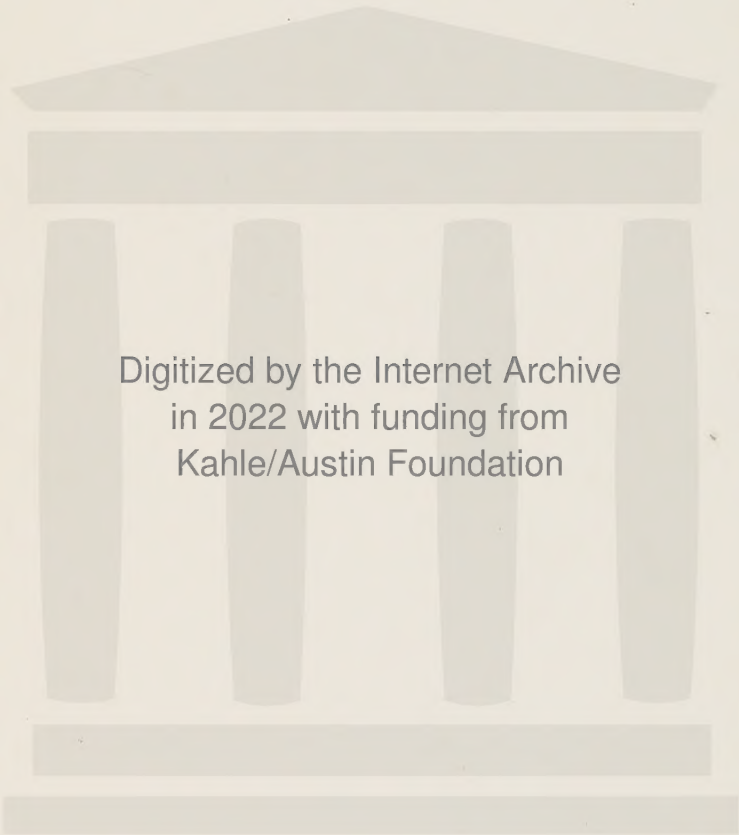
HOW TO SOLVE IT IN

LISP

PATRICK J. HALL

Σ
SIGMA
PRESS


Digitized by the Internet Archive
in 2022 with funding from
Kahle/Austin Foundation

https://archive.org/details/howtosolveitinli0000hall

How to Solve it in LISP

Patrick J. Hall

SIGMA PRESS
Wilmslow, United Kingdom

Copyright © P. J. Hall, 1989

All Rights Reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without prior written permission.

First published in 1989 by
Sigma Press 1 South Oak Lane, Wilmslow, SK9 6AR, England.

British Library Cataloguing in Publication Data
Hall, Patrick, 1949 -
How to Solve it in Lisp.
1. Computer systems. Programming languages:
Lisp language
I. Title
005.13'3

ISBN: 1-85058-005-7

Distributed by
John Wiley & Sons Ltd., Baffins Lane, Chichester, West Sussex, England.

Printed by Interprint Ltd, Malta

Cover design by Professional Graphics, Warrington, UK, based on "The Awakening Conscience" by Holman Hunt. Reproduced by permission of the Tate Gallery, London.

Acknowledgments: the author and publisher hereby acknowledge all trademarks and registered trademarks referred to in this book. In particular, we must thank Artificial Intelligence Ltd, who kindly provided a copy of Golden Common Lisp.

Preface

This book is intended to provide an introduction to the computer language LISP, long associated with Artificial Intelligence but equally suitable for a wide range of programming applications. No familiarity is assumed with LISP, or with any other language, and the book is aimed specifically at the beginner. Particular attention is paid to learning LISP via practical programming applications. A wide range of LISP functions is developed, including elementary statistics, time-series, input-output analysis, databases, queueing models, biological simulation, computer vision, machine learning and expert systems. In each case, minimal prior knowledge is required of the area under investigation. Full background information is provided for all programs. It is also hoped that the program commentaries are sufficiently detailed to help the reader understand the basic LISP concepts involved as the result of their practical role in the programs rather than as an abstract structure.

The dialect of the language employed is Golden Common LISP, running on an IBM PC computer or compatible. The explanatory nature of the text, however, should make the book also useful for those using other dialects.

This book could not have been attempted without the direct help, and implicit aid, of many people. My thanks go to Graham Beech, of Sigma Press, whose initial idea it was and whose constant encouragement, throughout has given me the confidence to continue. Tim Lewis, of Artificial Intelligence Limited, provided the evaluation copy of Golden Common LISP, as well as reading the manuscript. Electronequip and Arkrain Limited have both given technical assistance when required. For early encouragement I shall always feel indebted to my former colleague and co-author, John Scriven, as well as to David Lawrence, who first suggested that I should divert some of my attention from blackboard to word-processor! Mark Beard, Lucy Hopgood, Pauline Davis and others at the time of this conversion also deserve credit for their tacit support, or at least suspension of disbelief. Naturally, I am grateful to my mother and friends for excusing my lengthy evenings at the keyboard and apparent deafness to most other events taking place at the same time.

Patrick Hall

For Lucy Hopgood,
to acknowledge her transition from protegé to muse.

'All the good jobs are taken,
and I can do nothing but lisp in numbers'

(Aldous Huxley, *'Crome Yellow'*)

CONTENTS

1. A First Look at Lists: an Introduction to LISP 5
The History of LISP 5
The Role Played by Atoms 8
General Form of a LISP Function 9
Evaluation of an Atom 10
Evaluation of a List 12
The NULL List 15
Mixed Calculations in LISP 16
Creating New Lisp Functions 17
Summary 21

2. Choice and Repetition: Control Structure in LISP 23
Predicate Functions 27
Further Iterative Program Control 29
Discovering Perfect Numbers 36
Summary 40

3. Figuring It Out: Elementary Statistics 42
Simple Statistics - the Mean, Median and Mode 46
Top-down Programming, a Hierarchy of Functions 48
Arrays and Storing Data 57
A Bubble Sort in LISP - SORT-ARRAY 60
Scientific Notation in Golden Common LISP 64
Newton-Raphson Calculation of a Square Root 64
Measuring the Dispersion of Numbers 67
Summary 69

4.Minding Your Own Business...Time Series and Input-Output Analysis 70
Forecasting and Time Series 70
Analysing a Time Series 71
Screen Display 79
LISP and Input-Output Analysis 86
The Solution of Equations in LISP 90
Entering a Matrix in LISP 91
Calculating the Determinant 92
Finding the Inverse 94
Linear Equations in LISP 98
Summary 100

5. A La Recherché: Records, Trees and Networks 102
LISP Data Structure - The Property List 102
Using a Property List 105
A Practical Application of the Database 108
Networks and Trees 116

ARISTOTLE in Action - the Network Assembled 122
Summary 132

6. Minding Your Trees and Queues: Further Data Structures 134
Using Property Lists to Represent a Tree 136
Tree Structure and Search Techniques 139
Telling the Time 155
Simulating Arrival Times 161
Recursion in LISP 167
Summary 169

7. Life-like LISP: Biological Simulation
Simulating Cell Growth 176
BACTERIA's Principal Algorithm 177
Common LISP Primitive -SET
A LISP Petri Dish - DISH 180
Seeding the Dish with Microbe 183
Simulating a Biological Process 187
Summary 193

8. Seeing the Light: Computer Vision and Pattern Recognition
Pattern Recognition in LISP 194
Simulating Computer Vision - RETINA 196
Accountability in RETINA 205
Comparing with Known Shapes - RECOGNISE 209
Assembling RETINA's Component Functions 210
Summary 215

9. Knowledge to Order: Expert Systems
Representing Knowledge in an Expert System 216
Frames 217
Scripts 217
Semantic Networks 218
Production Rules 219
The Need for Machine Learning 221
The Importance of Feedback 222
A Simple Learning System 224
An Astronomical Application 241
Summary 246

Appendix A: Some A.I. Themes 247

Appendix B: References 264

Appendix C: Solutions 266

1

A First Look at Lists: an Introduction to LISP

The History of LISP

The birth of LISP is inextricably bound up with the origin of Artificial Intelligence and the legendary Dartmouth Conference of 1956. This was organised by John McCarthy, helped by Marvin Minsky, with the intention of studying AI. At the time this was nearly named 'Automata Studies' instead by McCarthy. At the conference McCarthy met Allen Newell and Herbert Simon, who had an early AI program, Logic Theorist, already working. To implement this program, they had specially written the very first list processing language, IPL. McCarthy was sufficiently impressed to begin the construction of what he hoped would be a better list processing language. Initially, he appeared more concerned with an ideal and theoretical, rather than a practical, language, but the involvement of his students encouraged him further. In particular, Steve Russell who wrote the evaluation function which turned the theoretical structure into a real programming language.

The traditional image of LISP is that of a language exclusively dedicated to problems in AI. Indeed it would be very foolish to pretend that LISP has not evolved as a tool specifically for it. The growth of AI is probably even the reason why LISP has survived as a viable language, despite its age. All the same, LISP can be used for a variety of applications which would not immediately seem to qualify as AI. It is one of the objectives of this book to show that this is the case. The earlier programs developed in the subsequent chapters certainly are not AI, although it is hoped that more 'intelligent' algorithms, in, for example, pattern recognition, do begin to control the later applications demonstrated.

How to read this book

It always seems impertinent to suggest to anybody how they should read a book. People's techniques vary enormously, whether of the 'skim and scan' variety or the dedicated 'memorise the ISBN number first' approach. At the same time it can help in understanding new material if a framework is established around which all the separate concepts can initially be arranged, before beginning to concentrate upon details.

The purpose of this book is to introduce LISP as easily as possible. No prior knowledge of the language is assumed. Neither is it necessary to have familiarity with any other

computer language. What is important is that the reader should test each new idea thoroughly at the keyboard as soon as possible before moving on to a later section. As suggested above, learning is usually achieved most quickly when poised on the verge of ignorance. It is the examples which do not give the expected responses which teach, because then ideas have to be reformed before they are finalised.

LISP is intended to be seen as a problem-solving language and it is constantly illustrated via realistic applications. Chapter Two continues the description of basic LISP, dealing with the control structures which will be needed in later programs, and illustrating them with examples drawn from number theory. Chapter Three introduces simple statistics and sorting methods. Chapter Four continues this theme into time-series and input-output analysis. There is thus a deliberate bias towards mathematics in LISP throughout the initial section of the text. This has the advantage of allowing LISP coding to be developed in a subject area which should be familiar for the majority of readers. Only fairly fundamental mathematical concepts are assumed. Additionally, with all the applications introduced, an account is first given of the essential background material. This applies to all programs developed in the book.

Chapter Five investigates the way in which information can be stored and retrieved in LISP and demonstrates this with different data structures. The material leads into the use of queueing theory, which is developed in Chapter Six into a detailed queueing model. Also in Chapter Six the concept of computer simulation is introduced and extended in Chapter Seven in a biological system. Chapter Eight deals with pattern recognition and machine learning in the context of simulated computer vision. Machine learning is continued in Chapter Nine and directed here towards expert systems.

All the functions developed are explained as fully as possible. The algorithms are described carefully and the actual LISP code teased apart into fine detail. It should be possible to modify the material given here and adjust it for other functions. With an explicit commentary provided for all the illustrative material, this will not be too difficult.

The structure of the book does imply that all the chapters will be read sequentially. LISP primitives are introduced only as required, and then illustrated via their role in the particular application being developed at that stage.

Introduction

The fundamental concept of LISP is the list. The name of the language is, indeed, an acronym for LISt Processing. The way in which lists dominate the whole of LISP can seem a little forbidding to a person approaching the language for the first time, as one taken in isolation may appear oddly vacuous and seem to imply that nothing of value could possibly be concealed inside. At the same time, even quite a modest piece of LISP code, with its neatly nested lists within lists, initially may be remarkably opaque and destroy self-confidence even before starting. The only way of tackling this problem is to suspend disbelief in the efficacy of lists and list processing long enough to see that

useful results do appear surprisingly quickly. The beginner has to accept that an apparently trivial combination of names and brackets could be a significant creation, representing hours of intellectual effort by some industrious LISP practitioner.
Before going any further it is important to appreciate precisely how a typical list is constructed. Here is an example.

```
(WELLS WILDE WOOLF WYNDHAM)
```

The first detail to note is that the list begins and ends with round brackets. These indicate the content and extent of the particular list being considered. All lists in LISP are delineated in this way. The next point is that there are precisely four elements in the list, separated from each other by spaces. Spaces and brackets are the principal separators for elements in a list. There is, incidentally, no need to have additional spaces at the beginning and end of any list. Some may feel that this appears neater to the human eye.

```
( WELLS WILDE WOOLF WYNDHAM )
```

However, it does not to the computer. The opening and closing spaces are redundant. If they are typed in, no error will occur but the IBM PC-compatible running Golden Common LISP will politely remove them.

Hyphens cannot be used to separate the elements and so the list:

```
(THE-INVISIBLE-MAN SALOME TO-THE-LIGHTHOUSE THE-
CHRYSALIDS)
```

also has just four elements and not nine. Fortunately, hyphens are quite a natural way of linking together items in ordinary language and so might not have been expected to separate the elements of the list. Rather unexpectedly, commas cannot be used to distinguish elements either. Golden Common LISP will not allow the user to enter an item like this:

```
(SUGAR,BUTTER,EGGS,FLOUR)
```

The vital factor in any list's structure is the nature of the elements themselves. These completely determine what the list does and how significant it is. Here, LISP again possesses considerable simplicity. An element can either be another list, or an item called an atom. This list:

```
(GOLDFISH (CORGI SPANIEL) (SIAMESE PERSIAN BURMESE))
```

consists of three elements. The first is the atom, `GOLDFISH`. The second is the list `(CORGI SPANIEL)` and the third is the list `(SIAMESE PERSIAN BURMESE)`. `(CORGI SPANIEL)` and `(SIAMESE PERSIAN BURMESE)` are regarded as sublists

of the original list and consist respectively of the further elements CORGI, SPANIEL, SIAMESE, PERSIAN and BURMESE.

A term often used to refer either to lists or atoms is 'symbolic-expression', shortened to 's-expression'.

Counting the Brackets

Care must be taken when one list is embedded as a sublist in another. It is essential that each list has an opening and a closing bracket and it can be a little taxing at first to make sure that this always happens when a number of lists are nested in a complicated hierarchy. For example, a perfectly valid list is given by:

```
((A B) C (D E (F G H)))
```

but not by the almost identical list:

```
((A B) C (D E (F G H))
```

In the first of these examples there are three elements in the initial list. These are the sublist (A B), the atom C and the sublist (D E (F G H)). This last element itself contains three elements, the atom D, the atom E and finally the further sublist (F G H). However, in the second example it can be seen that the third element is (D E (F G H). This is not acceptable as an item in LISP because it contains an unpaired opening bracket. With experience in scanning list structures, such errors are quickly noticed and probably as easily detected as spelling or punctuation mistakes are in everyday language. For the LISP novice, however, it is undoubtedly a good idea to develop the habit of moving outwards from a deeply embedded list within the overall structure, checking that it is balanced on both sides by equivalent opening and closing brackets. Alternatively, it may be preferable to start at the beginning of the list and read through it, making a mental tally of opening and closing brackets. A system, say, of 'scoring' +1 for an opening bracket and -1 for a closing should lead to an overall score of zero for a valid list.

The Role Played by Atoms

At this point it may be feared that some kind of infinite regression is afoot. An object called a list is being justified only in terms of other lists and a further, undefined item known as an atom. Clearly, it is vital to establish significance for the atoms in LISP, or else nothing meaningful is being established.

An atom may simply be a number. Here is a list in which all the atoms are numbers.

```
(270771 342433 662794)
```

Alternatively, an atom can be a symbol. Any collection of alphanumeric characters, naturally excluding just a series of digits, may be used as a symbol. Some typical examples of atoms being used as symbols could be `BOOKS-READ`, `FRIENDS` or `AVERAGE`. The names chosen can be used to suggest what the particular atom is intended to represent or the role which it performs. Symbolic atoms can be variables. They can also be function names. For example, `BOOKS-READ` could be a variable name representing the novels somebody claims to be familiar with, while `AVERAGE` could be a function which calculates the average of a series of numbers. LISP allows the user immense freedom in choosing symbolic atoms and in deciding what they are to do. This is, in fact, part of the attraction of the language, permitting a highly personal approach, with code tailor-made for specific applications. The next step is therefore to learn how to set up variables and use functions.

A function will be illustrated first because it will be easier afterwards to explain how variables are introduced in LISP if the basic concept of a function has already been explained. The particular function to be shown is a Golden Common LISP 'primitive', a piece of code already resident in the language when the user first comes to it, as opposed to the extra, personal additions to LISP which will soon be created.

Common LISP Primitive - +

The first example of a LISP function is the arithmetical operation of addition. In many LISP dialects, this function is called `PLUS`. However, in Common LISP, which is more competent in number-crunching, it is represented by the conventional sign of +. Here is this function in use.

```
(+ 2 7)
```

A point to deal with immediately is the mathematical form of the expression. LISP uses prefix notation, which means that the familiar 2 + 7 is replaced by + 2 7. At first it might seem a little unusual but there will be no difficulty in adjusting to this way of writing arithmetical commands.

General Form of a LISP Function

Of more significance is the way in which the function is placed first in a list, followed by the numbers upon which it is intended to operate. All LISP functions are employed in this way, occupying the head of a list in which their arguments (the information needed in order to operate the function) follow as the tail. This may be represented symbolically.

```
(FUNCTION ARGUMENT-1 ARGUMENT-2 ARGUMENT-3 ...)
```

As another example, `AVERAGE`, the function imagined earlier but not yet written, could be applied to a series of numbers like this:

```
(AVERAGE 7 3 2)
```

Naturally, there are many functions already resident in LISP when the user first comes to it. These functions can then be added to by new functions, built up from the existing ones by the LISP programmer. In this way, the language is adjusted to the particular problem being considered. A final LISP program will consist principally of various new LISP functions carefully designed to attain a desired result. The method by which such new functions are defined will be outlined shortly, but first a vitally important LISP function which allows the introduction of variables must be considered.

Common LISP Primitive - SETF

It will frequently be necessary to give a variable a chosen value. For example, in a BASIC program, a variable A can be set to the value 6 by the fairly intuitive command:

```
LET A = 6
```

Similarly, Pascal would require:

```
A := 6;
```

This process is called 'assignment'. It is an essential feature for a programming language to possess. LISP allows assignment in an analogous manner, using the function `SETF`. The function is employed in the standard LISP way. It forms the head of a list, followed by two arguments. The first of these is the atom used as the variable name. The second is the number being assigned to the variable. The equivalent assignment in LISP is therefore this list.

```
(SETF A 6)
```

Once this has been typed in, the atom `A` has the value 6. The term used in LISP is that 6 has been bound to the atom `A`. Before this was done, A was an unbound variable. The process of using the function `SETF` to assign a value to a variable is called variable binding. Similarly, this list uses variable binding to give the atom `B` the value `637`.

```
(SETF B 637)
```

`SETF` can be used to assign non-numeric values to atoms as well but before this is illustrated it is necessary to introduce the concept of evaluation.

Evaluation of an Atom

It is difficult for people to acquire computing skills unless they have 'hands-on' experience. This is much more than an overworked cliche. The very nature of an interactive computer system is that the usual constraint operating against a student, the

embarrassment of asking the teacher silly questions in front of her/his peers, no longer applies. A computer allows total idiocy indefinitely, responding kindly with patient error messages, until eventually a new concept has been acquired. Fortunately, in these circumstances, the number of errors dwindles rapidly and learning is achieved.

LISP on a personal computer, especially the Golden Common LISP package written for the IBM PC-compatible, has precisely this advantage. Learning can be very fast indeed. However, before turning to the keyboard and typing in some of the examples of simple LISP already given, it is necessary to find out a little about the LISP concept of evaluation.

First, it must again be stressed that LISP is an interactive language. The Golden Common LISP's 'listener' monitors keyboard entries the whole time, making appropriate responses of its own. It does not accept a large program all at once and then wait to be instructed to begin work on it. Instead it nibbles away constantly at everything it is told and readily queries anything which does not seem correct. This makes it ideal for learning. The constant analysis of the user's input is what is meant by the term evaluation.

When typing on a machine running Golden Common LISP the evaluation prompt will be present on the screen. This will be a single asterisk, followed by a space. Anything typed by the user appears after this. For example, if the atom `SAM` is entered the display will be:

```
* SAM
```

However, LISP's listener will expect an item it can understand to be typed at the prompt. Anything else will lead to an error message. Naturally, the user should be typing in either an atom or a list and this assumption is made when the machine evaluates the input. If the atom `SAM` has previously been bound to a value, `77`, this will be returned as soon as the return key, or the space bar, is pressed.

```
* SAM
77
*
```

After this value is displayed, the same evaluation prompt appears as before, awaiting the next entry from the user. Suppose, however, that the atom `SAM` has not been bound to a value. This is rather like expecting a person to talk about something with which he or she is totally unfamiliar and so the equivalent of 'I'm sorry but I don't understand you' appears on the screen. Evaluation will produce this display.

```
* SAM
ERROR:
Unbound variable: SAM
1>
```

The clarity of the error messages produced in Golden Common LISP makes progress for the LISP beginner much easier. Here, either the variable SAM has to be abandoned or else bound to a value. Note how the evaluation prompt itself has changed. The number, which is incremented each time a piece of LISP input has been evaluated to give an error, is a rough guide to how long the user has been sitting in front of the machine, or of how tired he or she is feeling! The original prompt can be restored by pressing simultaneously the three keys ALT, CTRL and DEL. This will return the prompt to the asterisk, or to the number appearing the last time that CTRL + Break has been pressed. The numbers indicate that deeper levels of the listener are being accessed.

Evaluation of a List

It is a little more complicated when a list is entered. Immediately the LISP listener examines the element at the head of the list, which it expects to be a function operating upon the arguments contained in the tail. Here is the display seen when the addition function is used correctly to add together two numbers:

```
* (+ 2 3)
5
*
```

In other words, if the first element is a recognised function, either one of the inbuilt primitive functions in Golden Common LISP or one that the user has defined (see section 1.4.4), then the appropriate action is taken. No error message should occur, unless, of course, a new function that has been defined has an inbuilt error of its own. The evaluation prompt reappears unchanged.

When evaluation encounters a list, it takes no action until the correct number of closing brackets have been typed by the user to balance the list. Then the evaluation takes place automatically, without any need for the user to press the return key. Even if the return key has been used several times during the entry of the list, no evaluation will occur until the final bracket is added. There is no reason why the addition illustrated above should not be typed on separate lines like this.

```
* (
+
2
3
)
5
*
```

This looks very untidy and confusing, but LISP evaluation has dealt with it sensibly. In fact, placing various components of a long list on separate lines like this is the only

way of clearly laying out new LISP functions in a readily understandable way. This will be seen in all the instances of function definitions which are encountered.

A problem does occur when the first element of the list typed in is not a function which LISP recognises. For example, as no function called AVERAGE has yet been written, an error will be reported if an attempt is made to employ it.

```
* (AVERAGE 4 5 6)
ERROR:
Undefined function: AVERAGE
while evaluating: (AVERAGE 4 5 6)
1>
```

Again it is reassuring how Golden Common LISP gives very detailed error messages. Here, AVERAGE has been detected as a function for which no definition yet exists. This is reported to the user and, in addition, the list in which it was found is stated. This proves invaluable when developing lengthy pieces of LISP code.

The READ-EVAL-PRINT Loop

Evaluation is taking place the whole time the IBM PC-compatible is running LISP. Everything typed in on the keyboard is read by the listener, then processed to give a value which is printed on the screen. This fundamental operation of LISP is sometimes referred to as the READ-EVAL-PRINT loop. It is important to realise that LISP is always looking for a value for the current keyboard input, which it then places on the screen. It can sometimes be disturbing for the LISP beginner to see these values appearing. Often they seem to have nothing to do with the process going on in the user's mind. Perhaps it might even appear that LISP and the programmer are striving for two different goals. LISP is determined to find a value, while the user is more concerned with the intermediate stages of the evaluation and with the display, almost incidental to the LISP evaluation process that appears on the screen before the final value. The only solution is to persevere!

Evaluation begins with the most deeply embedded sublists in the input presented. This is a very important point to bear in mind while trying to understand how a particular list is going to be handled by LISP. Again, it is only with practice that it becomes easy for the programmer to anticipate what will happen upon evaluation. Here is an example in which evaluation has a sublist to tackle first. The atom B is being bound to the value 8, but, in a slightly devious fashion, as the sum of 3 and 5:

```
* (SETF B (+ 3 5))
8
```

This is what LISP has needed to do. First, it has evaluated the sublist (+ 3 5) by looking at the head of the list and recognising the function + as one that it knows. It then applied the function to its arguments 3 and 5. Once the embedded sublist is

evaluated to give 8, LISP can then consider the outer list. At the head of this list is the function SETF and so the value 8 is bound to the atom B. All of this was carried out immediately, as soon as the list was balanced with its final closing bracket.

A final point to mention about evaluation in Golden Common LISP is that upper and lower-case characters are treated equivalently and so, in the example shown above (average 4 5 6) would have produced precisely the same error...

Inhibiting Evaluation in SETF

Now that the concept of evaluation has been firmly established, it is possible to extend the use of SETF to assigning non-numeric values to atoms. Suppose the atom SAM is to be bound to the non-numeric value MY-CAT. It might be thought that all that needs to be done is type:

```
* (SETF SAM MY-CAT)
```

Remember, though, how evaluation proceeds. The function SETF is seen at the head of the list. This tells LISP that the next element, SAM, is an atom to which a value is going to be assigned and that therefore it does not require evaluation itself. However, it does attempt to evaluate what follows. Just as in the example above, when the sublist (+ 3 5) was evaluated before LISP continued with the SETF, it now it attempts to evaluate MY-CAT:

```
* (SETF SAM MY-CAT)
ERROR:
Undefined function: MY-CAT
while evaluating: (SETF SAM MY-
CAT)
1>
```

Clearly, LISP's insistence upon constantly analysing its input can go a little too far. Evidently, some method is needed to prevent evaluation when some new piece of information has to be added to the existing LISP environment. Here, MY-CAT is simply MY-CAT. It does not possess any deeper level of meaning. LISP has to be ordered to accept it just as it stands.

There is a simple method for inhibiting the evaluation process. A single quote sign placed before an atom, or immediately outside the opening bracket of a list, will inform LISP that the item that follows must not be evaluated. There must be no space after the quote sign. This is how the atom SAM could have been given the value required:

```
* (SETF SAM 'MY-CAT)
MY-CAT
```

Alternatively, a more detailed description of SAM could have been assigned by quoting a list.

```
* (SETF SAM '(A BLACK AND WHITE
KITTEN))
(A BLACK AND WHITE KITTEN)
```

It may be wondered why it was not necessary to place a quote sign before the number in examples like that on the following page:

```
* (SETF C 89)
89
```

What has happened here, though, is that the number 89 has, indeed, been evaluated. Numbers simply evaluate to their obvious value.

The NULL List

There is one list in LISP which contains no elements at all. It can be written:

```
()
```

It is called the NULL, or the empty, list and it can also be represented by the atom NIL. NIL also has another role in LISP, as will be seen in the next chapter, and plays a very important part in the development of much of the language.

Common LISP Primitives - -, *, /, 1+, 1-

An example has already been given of an arithmetical function in LISP, the + function which adds numbers. This function does not have to be restricted to a single number pair and so a list with several numbers is also acceptable.

```
* (+ 4 9 12)
25
```

Here, all the numbers in the list have been added together. The three other basic arithmetic operations exist in LISP as well. In each case, the prefix convention is again followed for the position taken by the sign denoting the operation. This, of course, matches the LISP convention of having the function name at the head of the appropriate list. Thus subtraction is carried out in the same general way as addition, using the Common LISP function -. The name MINUS is used in some other LISP dialects.

```
* (- 67 45)
22
```

Here, the function has given the value 22 upon evaluation. Similarly, multiplication and division can be handled by the functions * and /. TIMES and QUOTIENT are the function names employed in other LISPs.

```
* (* 15 6 )
90
* (/ 78 5)
15.6
```

In each of these examples, care has been taken to ensure a space exists between the sign of the function and the first number. Failure to check this would result in an error message appearing. The listener would fail to distinguish between the function and the first argument and assume that a new, and unknown function, was being employed. Here, the answer would not be 8:

```
* (/24 3)
ERROR:
Undefined function: /24
while evaluating: (/24 3)
1>
```

This is quite an easy error to fall into, especially if typing too fast!

Often it is useful to be able to increase or to decrease the value of a variable by just 1. This could, of course, be achieved by typing in either of the two expressions:

```
* (+ 1 6)
7

* (- 1 6)
5
```

Alternatively, the slightly more compact 1+ and 1- functions can be used:

```
* (1+ 6)
7

* (1- 6)
5
```

Mixed Calculations in LISP

Golden Common LISP is not restricted to integer arithmetic in the way that some more limited dialects of LISP are. Mixed calculations are possible, although care must be taken to examine the order in which the relevant lists are nested. As before, the guiding principle is that the evaluation function works from the lowest level of brackets

outwards. This means that the programmer needs to consider which parts of the calculation need to be performed first and then make sure that the sublists associated with these calculations are the most deeply nested within the overall list. Here is a mixed calculation in ordinary arithmetic. Remember that these are now standard arithmetical brackets, not the specialised LISP ones!

```
( 2 + 5 ) * ( 12 / 3 )
```

2 and 5 are added to give 7. 12 is then divided by 3 to become 4. Finally, 7 and 4 and multiplied and the answer to the whole calculation is 28. Each of these individual steps must be taken into account when constructing the LISP list which will perform the same calculation. The LISP expression will be:

```
(* (+ 2 5) (/ 12 3))
```

When this list is evaluated, the embedded sublist `(+ 2 5)` is evaluated first to give `7`. Then the list `(/ 12 3)` evaluates to `4`. When both embedded lists have been dealt with, the function `*` at the head of the main list receives the values `7` and `4` and yields the final evaluation of `28`.

Creating New LISP Functions

The real power of LISP can only be experienced when the user begins to define new functions appropriate to her/his particular programming task. LISP possesses an extremely easy method of doing this. It can be illustrated initially by writing the function referred to above, `AVERAGE`. In this first version of the function, called `AVERAGE-A`, in contrast with a second function written later, three numbers will be accepted and their average value calculated. The second version, developed further on, will be defined to be more general in its application.

Common LISP primitive - DEFUN

The function used to define new functions in LISP is `DEFUN`. As with all LISP functions, it takes the first place in the list in which it is used. This is followed by the name chosen for the function, a sublist of the arguments needed and then finally the nested sublists necessary to perform the required task. Schematically, `DEFUN` is used like this:

```
(DEFUN Function name (List of arguments)
 (Necessary code))
```

Indentation has been used to emphasise the structure. The operation of the LISP primitive `DEFUN` in creating a new function can be demonstrated as the definition of `AVERAGE-A` is examined.

A New LISP Function - AVERAGE-A

Here is the new function, written using only existing LISP primitives.

```
(DEFUN AVERAGE-A (A B C)
 (/(+ A B C) 3)
```

The very first item encountered by the LISP listener is DEFUN. This indicates that what follows is a function definition. After this is the name to be employed when the new function is used. The third element in the list is an embedded sublist giving the three numbers for which AVERAGE-A will calculate an average value. This list of arguments is called the 'parameter list'. Finally, at the end of the list is a sublist, which actually calculates the average of the three numbers. This list is the symbolic-expression (/(+ A B C) 3). The mixed calculation is evaluated just as the earlier examples were. The embedded sublist, (+ A B C), adds the three numbers. After this calculation is completed, the answer is divided by three in the main list to give the average value. Finally, there is an extra closing bracket at the end of the function definition. This is paired with the initial opening bracket before DEFUN. As with all other legal LISP expressions, the brackets must be balanced.

AVERAGE-A can now be used as part of the LISP that the computer understands. It has been added to the LISP environment, as shown in this example.

```
* (AVERAGE-A 7 23 18)
16.0
```

Squaring a Number - SQUARE

An example is now given of another function definition, SQUARE, which calculates the square of any number given to it. The overall format of the function definition follows precisely the pattern already described. This new function is more significant than the example given above, because it will actually be employed in other new functions written to perform specific applications, which are developed later in the text.

```
(DEFUN SQUARE (N)
 (* N N))
```

Only one variable needs to be present on the parameter list, N. In the body of the function it is multiplied by itself in the symbolic-expression (* N N). As this is the last part of the overall list to be evaluated, the square of N is the value which appears on the screen. After the expression is the final closing parenthesis.

SQUARE can now be tested.

```
* (SQUARE 5)
25
```

Converting Temperature - C-TO-F

As a last example of a function definition, here is C-TO-F. It is a function which converts Celsius temperature to Fahrenheit. This is achieved by multiplying the Celsius temperature by 9, dividing by 5 and then adding 32. These separate calculations have to be reflected in the ordering of the embedded sublists in the body of the function.

```
(DEFUN C-TO-F (TEMP)
 (+ (/ (* TEMP 9) 5) 32))
```

The most deeply embedded list is the expression (* TEMP 9). This multiplies the temperature entered by 9. Around this list is constructed the additional code to divide this answer by 5. At this stage the expression is (/ (* TEMP 9) 5). Finally, 32 is added in the overall list (+ (/ (* TEMP 9) 5) 32). A final closing bracket ends the definition by balancing the bracket in front of DEFUN. Note how the variable name chosen for the parameter list is TEMP rather than T. The reason for this will appear in the next chapter.

Common LISP Primitive - READ

Few computer programs will be expected to operate in total isolation. They must be able to communicate with the person running them. It is inevitable that information will need to be typed into a program and further information obtained from it and displayed on the screen. Therefore, like other languages, LISP possesses input and output functions.

Input has taken place in the functions shown so far by explicitly including variables inside the list making the function call. More generally, it is achieved with the function READ. This obtains the last item typed in at the keyboard immediately before the return key, or space bar, is pressed. READ is obviously a function which does not require any arguments, since it cannot anticipate what might be typed next. It is therefore a little unusual compared with the previous LISP functions encountered, because it is used as a single element list. For example, typing the single word:

```
* (READ)
```

at the keyboard will cause the listener to postpone any further action until the space bar or return key is pressed. The value of READ is then whatever has been typed. Because READ also detects a space as well as the return key, it will only accept single items as its value.

The easiest way to employ READ is together with SETF in an embedded sublist. SETF is used to bind any keyboard entry which READ returns to a chosen variable. This expression might be typed.

```
* (SETF INPUT (READ))
```

When this is entered into Golden Common LISP and followed by the return key or space bar, the atom INPUT will acquire as its value whatever has been added after the expression. The display will appear like this.

```
* (SETF INPUT (READ))IBM-PC
IBM-PC
```

The first line may be confusing at first. LISP has accepted the balanced list (SETF INPUT (READ)) and immediately begun to evaluate it. First, it looks at the most deeply embedded sublist (READ), and realises that the return key or space bar must follow before evaluation can take place. Hence nothing at all will happen until the additional entry is made on the same line, IBM-PC, followed, of course, by the return key or space bar. Then IBM-PC becomes the value of (READ) and is bound to INPUT. It is also returned, as can be seen, as the value of the whole list. To check that INPUT has been bound as intended, simply type:

```
* INPUT
IBM-PC
```

INPUT evaluates, as it should, to its bound value of IBM-PC.

Common LISP Primitive - PRINT

Specifying an output function is sometimes unnecessary in LISP because the constant evaluation of all lists entered can provide precisely the information needed. More frequently, though, the display presented needs to clarify what is happening in the program. An output function is then essential. For this LISP uses the primitive function PRINT to place information on the monitor. Unlike READ, it will be followed in its list by a second item, the one which it displays:

```
* (PRINT '(THE DOVE DESCENDING BREAKS
THE AIR))
(THE DOVE DESCENDING BREAKS THE AIR)
(THE DOVE DESCENDING BREAKS THE AIR)
```

The repeat occurs because after PRINT places its argument on the screen, evaluation also takes place of this list. The value obtained, which naturally will be displayed, is the same as the initial output. Hence duplication occurs. In practice, this proves to be less annoying than it appears here, because in most functions some further item will be evaluated after any use of PRINT. It is, of course, the last item evaluated which has its value displayed and so the display of the PRINT's evaluation will be prevented.

Common LISP Primitive - EXIT

It is quite possible that at the end of a session with Golden Common LISP the user might want to run some favourite program on the IBM PC-compatible before switching off. To allow other software to be run when desired, the command EXIT will abandon the LISP environment and return the MS-DOS A> prompt to the screen.

```
* (EXIT)
A>
```

Summary

Before reading the next section, it is probably wise to look back and check on the major points covered so far. In particular, you should be prepared to accept the following.

1. LISP is a computer language in which the essential data structure is a list.

2. Lists consist of a series of elements separated by spaces and enclosed between a pair of round brackets.

3. The elements in a list may be further, embedded sublists or single items known as atoms. When sublists are included in a list, it is vital that all the opening and closing brackets match in pairs.

4. Atoms give meaning to the list of which they are elements.

5. An atom may be a number or a symbol. If it is a symbol, then it can either represent a LISP function or a variable.

6. When a function is used in LISP, it always occurs as the first element of a list which also contains the arguments on which the function will operate.

7. A value can be given to a variable by means of the LISP function `SETF`. This is known as variable binding.

8. LISP evaluates all keyboard entries to give a value to everything typed by the user.

9. Evaluation is inhibited by the use of a single quote. This allows non-numeric values to be assigned to atoms.

10. Numbers automatically evaluate to their expected value.

11. LISP can deal perfectly adequately with arithmetic.

12. LISP already contains a large number of essential functions as part of the primitives present when the user first enters the LISP environment. However, the real power of

the language is only achieved when the programmer adds specific functions of his/her own to this core.

13. New functions are defined using `DEFUN`.

14. LISP contains the input/output functions `READ` and `PRINT`.

15. The Golden Common LISP environment is left via `EXIT`.

Exercise 1

Decide which of the following are valid lists.

1. `(BACH ((BARTOK) ((BEETHOVEN) BOULEZ) (BRAHMS)) BRITTEN)`

2. `(GERSHWIN (GLASS GLUCK GOTTSCHALK GRAINGER (GRIEG)) GRUBER)`

3. `((HANDEL ((HARRISON-BIRTWISTLE) ((HAYDN) HENZE))) HINDEMITH)`

4. `(((MAHLER) (MAXWELL-DAVIES) (MEDELSSOHN))) MOZART)`

5. `((WAGNER) (WALTON) (WEBER (WEBERN WEILL WOLF))`

Count the number of elements in each of these lists.

6. `(WATER SUGAR CITRIC-ACID ANTI-OXIDANT E104 E110 E150)`

7. `(EINSTEIN-ON-THE-BEACH SATYAGRAHA AKHNATEN)`

Devise appropriate LISP expressions to perform each of the following calculations, shown here in conventional notation. State what the value of each will be when evaluated.

8. `(23 * 5) + (13 - 6)`

9. `(4 + 6) * (25 / 5)`

10. `(12 + 2) * (7 / (8 - 3))`

11. Write a function which cubes a number.

2

Choice and Repetition: Control Structure in LISP

Introduction

In the construction of even the simplest algorithms, two fundamental features emerge. Consider the problem of ironing your shirt, as depicted in the very basic flowchart in Figure 2.1.

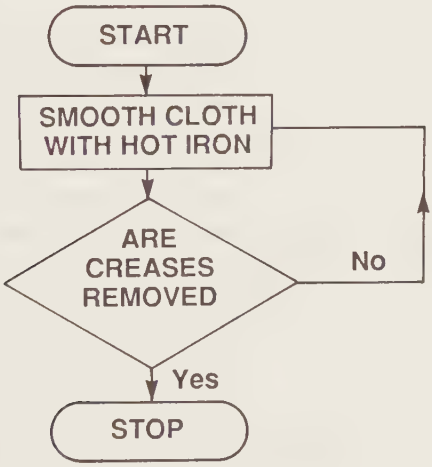


Figure 2.1

You apply the iron. You check if there are creases. You apply the iron a second time. Again you make a decision about whether the shirt is sufficiently crisp and smooth to the touch. The whole process consists of action being repeated or not, depending upon the outcome of a decision. Similarly, there are two essential processes in the writing of any significant computer program. The option should be available of specifying that particular sections of program code are repeated more than once. In addition the possibility must exist that the operation of the program can move into some new direction according to an intermediate result reached. Looping and branching are vital to program structure. A third possibility for control structure, recursion, will be investigated in a later chapter.

The distinctive way in which Golden Common LISP can handle these vital aspects of program construction will be examined in this chapter, beginning with the DOTIMES loop.

Common LISP Primitive - DOTIMES

A DOTIMES loop is one way of carrying out the same set of instructions more than once in order to achieve some desired result. This method of constructing a program loop can only be used when it is known in advance how many times the section of program is to be performed. It is not designed to loop indefinitely until some specific exit condition is met. For those familiar with BASIC programming, a DOTIMES loop is therefore similar to a simple FOR-NEXT loop, rather than a REPEAT-UNTIL or WEND-WHILE structure.

The structure of a DOTIMES loop is:

```
(DOTIMES (Variable / Number-of-repeats / Value)
         (Content of loop))
```

Some explanation is needed. The first element in the list is, as expected, the name of the function itself. Remember that on evaluating any list LISP looks immediately at the first element to discover what the list is meant to be. Here, it realises that a DOTIMES loop is requested. After this is a sublist consisting of three elements. These are a variable name, the number of times the loop has to repeat, and the value which the DOTIMES loop should present upon evaluation. Finally, within the main list is the loop body. This is the part the programmer is concerned with because here is contained the piece of LISP code that the DOTIMES loop is intended to repeat.

Any legal LISP atom can be used as the variable name for the loop. It will only have a value within the loop itself. Requesting it outside the loop will lead to an 'unbound variable' error. The value presented by the loop can be selected according to the particular application. Often it is convenient to have the loop simply returning T. The operation of a DOTIMES loop will be much more readily understood if a simple example is investigated. Suppose this has been typed in on the IBM PC-compatible:

```
* (DOTIMES (X 5 'Finished))
```

Note that no loop body has been enclosed in the DOTIMES expression. This is quite acceptable and the embedded sublist will still select the atom X as the variable for the loop, assigning to it, in sequence, five values. These always begin at zero and increment by one each time. So the values 0 1, 2, 3, 4 are given to X as the loop is performed. After this the DOTIMES loop will have its value displayed on the screen. This is the last element in the sublist 'Finished, which, of course, could have been chosen to be any message or LISP item capable of evaluation. Note that it has been quoted because otherwise LISP would attempt further evaluation and an error would be given. All that will appear on the screen, however, is this.

```
* (DOTIMES (X 5 'Finished))
Finished
```

This will probably seem unconvincing. There is no proof at all that anything has been repeated in the loop. It is not even clear whether the value of X has been incremented as expected, before becoming unbound again. In order to check that LISP has been working during the execution of the loop, a modified version can be tried, written as a test function TESTA. Here, the usual indentation for a DOTIMES loop is also shown.

```
(DEFUN TESTA (N)
 (DOTIMES (X N 'Finished)
          (PRINT X)))
```

Two principal alterations have been made. The loop can now be made to repeat a given number of times, determined by the value of N on the parameter list. Additionally, the value of the loop variable, X, is printed at each cycle and so the way in which it increments during the execution of the loop can be witnessed. The function will operate like this, choosing 5 again as the number of cycles.

```
* (TESTA 5)
0
1
2
3
4
Finished
```

Doubts about whether X was altering in value should be assuaged by the proof given on the screen. When a DOTIMES loop is used seriously, rather than in a trivial example such as this one, the value adopted by the loop variable can be incorporated in a significant way into the associated algorithms. Many examples of this will be shown in the functions developed later in this text.

An Improved Average Function - AVERAGE-B

A DOTIMES loop can now be employed to help redefine the function AVERAGE-A. The initial previous version only found the value of three numbers, which had to be typed explicitly as part of the function call. Here, a generalised function allows the user to select how many numbers are to be averaged.

```
(DEFUN AVERAGE-B NIL
 (PRINT '(How many numbers?))
 (SETF A (READ))
 (SETF B 0)
 (DOTIMES (X A T)
          (SETF B (+ B (READ))))
 (PRINT '(Average is))
 (PRINT (/ B A)))
```

AVERAGE-B now has NIL as its parameter list because the numbers it operates upon will be accessed directly from the keyboard via READ. A prompt is placed upon the screen to request how many numbers are to be typed. SETF then binds this to the atom A, which is used in the DOTIMES loop to determine how many times the loop repeats. The atom B is initialised as zero by (SETF B 0) and then continually updated inside the loop with the numbers typed. This is done by the expression (SETF B (+ B (READ))). A fourth closing bracket appears here to mark the end of the loop. The value returned by the DOTIMES loop is T. This is fairly unimportant, since it will not be placed on the screen. It is always the last evaluation which actually appears in LISP. PRINT is used to show the message, (Average is). Finally, the average is calculated by dividing B by A, the number of keyboard entries being averaged. This is the final value found when the overall function is evaluated and so is the number which is displayed, by the expression (PRINT (/ B A)). The third bracket is the closing parenthesis of the function definition. Here is AVERAGE-B being used to calculate the average of four numbers:

```
* (AVERAGE-B)
(How many numbers?) 4
3.4
67.1
7.42
1.9
(Average is)
19.955
```

Now that one form of program looping in LISP has been introduced, the next feature of control structure which needs to be illustrated is conditional branching. However, before this two important LISP concepts must be discussed, the Boolean values of T and NIL and predicate functions.

Boolean Values - T, NIL

The logical concepts of true and false, featuring in Boolean algebra, also perform an important, and a very practical role in LISP programming. True is represented by the atom T and false by NIL. NIL is equivalent to the empty list. This can be seen by typing:

```
* ()
NIL
```

In practice, any non-NIL value is equivalent to T, as will be discovered with experience. It might appear confusing at first to have T and NIL standing for true and false and it needs to be appreciated that this is just a simple convention which, with constant use of LISP, becomes automatic in the programmer's thinking. In particular, the use of predicate functions soon produces an intuitive feel for T and NIL.

Predicate Functions

The conditions required to allow branching in LISP are provided by special functions called predicates. There are a large number of predicates contained among the Golden Common LISP primitives and in developing effective LISP programs the user soon learns to write new ones. A predicate is a function which looks at a particular LISP expression and decides whether it is true or not. It then returns a value of `T` or `NIL`. This is best illustrated by considering a few of the particular predicate functions available in Golden Common LISP. The first of these ends in the letter `P`, which is characteristic of many, though not all, predicate functions.

Common LISP Primitive - INTEGERP

This function takes a single number as its argument and returns `T` if it is an integer and `NIL` otherwise.

```
* (INTEGERP 7)
T
* (INTEGERP 7.4)
NIL
```

Common LISP Primitives - >, <, =

The standard signs of inequality and equality exist as mathematical predicate functions in Golden Common LISP, with their conventional interpretation remaining intact. The 'greater than sign', when used as a predicate function, will therefore return T if the first of its arguments is greater than the second and `NIL` if it is not.

```
* (> 89 2)
T
* (> 3 25)
NIL
* (> 15 15)
NIL
```

In some other dialects of LISP, this predicate function is called `GREATERP`, a name which emphasises its role. If this was considered a desirable feature of the LISP environment, `GREATERP` could simply be added via a new function definition:

```
(DEFUN GREATERP (A B)
 (> A B))
```

The 'less than' sign also performs as a predicate function:

```
* (< 5 6)
T
* (< 6 5)
NIL
* (< 5 5)
NIL
```

The function LESSP present in some LISPs can again be emulated, if desired, with a new definition:

```
(DEFUN LESSP (A B)
 (< A B))
```

The equality of two numbers can be tested with the mathematical predicate function, =.

```
* (= 42 42)
T
* (= 67 78)
NIL
```

Common LISP Primitive - IF

The Boolean values of T and NIL, as embodied in the operation of predicate functions, can now be added to LISP's structure in order to allow branching to be included in programs. A general form of a conditional branch can be expressed in ordinary language as:

IF (Some condition) THEN (An action) ELSE (Another action)

This is a symbolic way of expressing a statement like 'If Hilary comes to my party I will be overjoyed, but if she doesn't I shall drink heavily in the corner'. In a similar way, this structure is present in 'If it rains the picnic will be cancelled, but otherwise we'll take sandwiches to the beach'. Computer languages need to formalise this logical structure in their syntax. The form used by LISP for this condition involves a new function, IF, which is placed at the head of a conditional expression. The general form is:

(IF (Condition) (First action) (Second action))

Naturally, this is a list with the function appearing first. Three sublists follow. In the first, a condition has to be written in suitable LISP code. Then are the two possible actions for LISP to select. If the condition proves true, the first of these actions is chosen, but if it is false, then the second action is performed. The similarity with the way in which conditions are articulated in common discourse should be evident.

Another trivial function can be written which illustrates the way in which IF operates.

```
(DEFUN CONDITION NIL
 (PRINT '(Please type a number))
 (SETF N (READ))
 (IF (= N 21)
  (PRINT '(That is only half the
  answer))
   (PRINT '(Consult the Hitch Hiker's
   Guide))))
```

Although this function does not really do anything, apart from demonstrating the way in which IF works, it does at least provide another useful mental test of counting how many final brackets are needed. Here, the first ends the second list which is printed. The second bracket is the final parenthesis of the PRINT itself, which is the second action of the IF condition. The third marks the limit of the conditional expression, while the fourth bracket simply finishes the function definition. Also note the way the conditional expression has been indented an extra place on each fresh line it occupies. Generally, such a format makes it easier to understand pieces of LISP code and will be followed from now on.

CONDITION can now be tested.

```
* (CONDITION)
(Please type a number) 56
(Consult the Hitch Hiker's Guide)
NIL
* (CONDITION)
(Please type a number) 21
(That is only half the answer)
NIL
```

The two different actions have taken place precisely as expected from the structure of the conditional and the given input. Despite the superficiality of this initial example, the use of IF in subsequent program code will be extensive.

Further Iterative Program Control

A DOTIMES loop is limited in its application because it cycles a fixed number of times, declared at the beginning of the expression. The repetition is unconditional and will not be affected by anything which may occur within the loop body. Fortunately, a more generalised iterative structure exists in Golden Common LISP. This is the DO loop, which does permit conditions to control the number of repeats and is therefore far more versatile in its use.

Common LISP Primitive - DO

The structure of a DO loop is:

```
(DO ((Variable / Initial value / Variable step))
    ((Exit condition))
    (Content of loop))
```

The important additional feature of the DO loop is the presence of the conditional expression before the loop body. When this evaluates as T, the loop is ended immediately and attention is given to whatever piece of program code follows after the loop.

The double brackets allow for the inclusion of more than one variable within the loop. The operation of DO loops can only really be understood once practical use is made of them. However, a series of test functions can help indicate general principles. The first of these is TESTB. Note the characteristic indenting used to emphasise the body of the loop.

```
(DEFUN TESTB (A B C)
 (DO ((X A (+ X B)))
     ((> X C))
     (PRINT X)))
```

This function requires three values to be typed when it is employed. The first will be the initial value of the loop variable X. The second is the amount by which X is increased at each cycle of the loop. The third is the upper limit for X which controls the exit condition. As soon as X passes this value the loop ends. In order to show what is happening, the current value of X is displayed by (PRINT X) at each loop cycle. Here is TESTB used with three numbers selected as its arguments.

```
* (TESTB 5 1 11)
5
6
7
8
9
10
11
NIL
```

As expected, X begins with the value 5 and then increases by 1 each time the loop is executed. The final value reached by X is 11. The next value of 12 causes the loop to end at the condition (> X C). Again, here is TESTB with different initial values controlling the loop.

```
* (TESTB 8 3 25)
8
11
14
17
20
23
NIL
```

X now begins at 8 and increments in threes. The final value it reaches is 26 but this is never printed to the screen because the condition (> X 25) causes the loop to end.

One DO loop can be nested inside another, as shown in this further test function, TESTC.

```
(DEFUN TESTC (A B C D E F)
 (DO ((X A (+ X B)))
     ((> X C))
     (PRINT X)
     (DO ((Y D (+ Y E)))
         ((> Y F))
         (PRINT Y))))
```

Six variables appear on the parameter list for TESTC. The first three control the outer DO loop and the remaining three govern the operation of the inner loop. X begins at the value A and increments by B at each cycle of the outer loop. This loop ends when X exceeds C. The value of X for each cycle is printed and then the inner loop completes an entire cycle before the next execution of the outer loop. Here, Y begins with the value D and is incremented by E before the inner loop ends when the value of F is exceeded. All of these stages can be seen occurring in this typical execution of TESTC.

```
* (TESTC 10 1 12 1 2 6)
10
1
3
5
11
1
3
5
12
1
3
5
NIL
```

The values of X that appear are 10, 11 and 12 but between these occur each time the whole cycle of Y values, 1, 3 and 5.

An example can also be given of a two variable DO loop. Here, no nesting is taking place. The variable X begins with a value A and increments by B. The second variable Y begins as C and increments by D. The loop ends when X is greater than E.

```
(DEFUN TESTD (A B C D E)
 (DO ((X A (+ X B))
      (Y C (+ Y D)))
     ((> X E))
     (PRINT X)
     (PRINT Y)))
```

The difference between this two variable loop and the previous two nested loops can be seen in an example.

```
* (TESTD 20 5 100 1 37)
20
100
25
101
30
102
35
103
NIL
```

A DO loop can now be used for the first time in a new function definition.

Finding an Integral Value - INT

An extremely useful function to possess is one which ignores the decimal part of any number typed in and returns just a whole number. A very simple algorithm can be used to create this function. Suppose the initial number is 5.4. One is repeatedly subtracted until a negative result is produced. This yields the series of numbers 4.4, 3.4, 2.4, 1.4, 0.4 and finally -0.6. If this last value is subtracted from the original number, this gives 5.4 - (-0.6) or 6. This answer is just one more than the integral value required, which can therefore be obtained by subtracting a further unit. All of this can be coded quite easily as a LISP function.

```
(DEFUN INT (N)
 (SETF NA N)
 (DO ((X 1 (1+ X)))
     ((< N 0))
     (SETF N (1- N)))
 (- NA N 1))
```

The number from which the integral value is to be obtained is placed on the parameter list. The same value is also given to the variable NA. In the DO loop which follows the value of the variable, X does not enter explicitly into the loop body. Instead, the variable N is repeatedly reduced by 1 until it is negative, as determined by the condition (< N 0). Outside the loop this is then subtracted from the original number, still stored as NA, and the additional 1 removed as well, by the expression (- NA N 1). As this is the last part of the function to be evaluated, its value is the one returned. As seen above, this is the integral value required. Note how the subtraction function has here accepted more than two numbers.

The function operates as in this illustration.

```
* (INT 5.6)
5.0
* (INT 12.78)
12.0
* (INT 0.8)
0.0
```

The DO loop control structure is used again in the next function to be defined.

Removing a Trailing Zero - STRIP-ZERO

This function takes an integer in the form 6.0 and returns it simply as the integer without the decimal point or trailing zero 6. This will be used quite frequently in later function definitions because often a value being passed as a control variable from a calculation to a loop will possess an unacceptable trailing zero.

```
(DEFUN STRIP-ZERO (N)
 (DO ((X 0 (1+ X)))
     ((= N X))
     (SETF WN X))
 (1+ WN))
```

In the DO loop, the 'whole number' variable WN takes the value of the loop variable X at each pass. The loop exits when X reaches the value of the number N. WN then has to be incremented once more outside the loop to equal N, without the trailing zero.
The function is illustrated in this example.

```
* (STRIP-ZERO 4.0)
4
```

An Alternative Integral Function - INTEGER

The last two functions can be combined to give INTEGER, which returns integral values without the decimal place. In many applications, this version of the function will be preferable.

```
(DEFUN INTEGER (N)
 (SETF NA N)
 (DO ((X 1 (1+ X)))
     ((< N 0))
     (SETF N (1- N)))
 (STRIP-ZERO (- NA N 1)))
```

It gives results of this form:

```
* (INTEGER 5.23)
5
```

A Factor Predicate Function - FACTORP

Another useful function to define is `FACTORP`. This takes two arguments and returns `T` if the first is a factor of the second. If it is not, the function evaluates to `NIL`. It therefore behaves as a predicate. `FACTORP` is coded as follows.

```
(DEFUN FACTORP (A B)
 (= (/ B A) (INT (/ B A))))
```

Its structure is quite simple, the body of the function consisting of just one symbolic expression. This is the mathematical predicate of equality, but it is made more complicated by both of its arguments being expressions in themselves. This, of course, is standard LISP procedure, since evaluation takes place at the lowest level first. The two arguments will therefore be fully evaluated before the predicate function in which they are embedded is evaluated in turn.

If `A` is a factor of `B`, it will divide into `B` without a remainder. There will be no difference between `(/ B A)` and its integral value. This is tested by `(= (/ B A) (INT (/ B A)))`, which will return `T` only if `A` is a factor and `NIL` otherwise. This can be seen in these examples.

```
* (FACTORP 2 46)
T
* (FACTORP 4 20)
T
* (FACTORP 7 20)
NIL
```

A Prime Number Predicate Function - PRIMEP

`FACTORP` can be used in a predicate function designed to determine prime numbers, `PRIMEP`, coded as follows.

```
(DEFUN PRIMEP (N)
 (SETF M 2)
 (DO ((X 2 (1+ X)))
     ((FACTORP X N))
     (SETF M (1+ X)))
 (= M N))
```

In this function, the values assumed by the variable in the DO loop play a vital part in the algorithm employed. The loop repeats until the variable X divides into the number under test, N. This is discovered by the exit condition (FACTORP X N). When X does divide into N, this expression evaluates as T. Because of the steady increase in the value of X, inevitably it will eventually divide exactly into N, possibly when X actually equals N. At whatever stage the loop exits, the variable M will have the same value as X. This is because although the loop is left at the beginning of the cycle, M is made equal to one more than the value of X in the previous cycle. Only in the case of prime numbers will M then be equal to N. The final predicate expression, (= M N), will thus return T or NIL in an appropriate fashion. The loop has to begin with X = 2, and M is similarly initialised as 2, because all numbers divide by 1 and the definition of prime numbers must exclude this vacuous case.

Detecting Prime Numbers - PRIMES-TO

A function which locates prime numbers can now be created using the newly defined predicate function PRIMEP. The name chosen for the function is PRIMES-TO. In typical LISP fashion, this has been selected to indicate clearly what the function will do. It is called with a single argument. This is the integer up to which the function will search for prime numbers. A typical call to the function will therefore be (PRIMES-TO 20). Apart from the brackets, which might still seem a little strange, this is almost natural English. It shows how conveniently self-explanatory LISP coding can be made to appear.

PRIMES-TO is coded like this.

```
(DEFUN PRIMES-TO (N)
 (PRINT 1)
 (DO ((X 2 (1+ X)))
     ((= X N))
     (IF (PRIMEP X) (PRINT X) T)))
```

The function begins with the primitive which tells Golden Common LISP that a new function is being written. This is followed by the name selected for the function PRIMES-TO, and the parameter list (N). Located here is the variable which will store the integral value marking the upper limit for testing numbers.

The next part of PRIMES-TO could be described as a 'fudge factor'! It is needed because the algorithm which tests numbers to decide whether they are prime or

composite will not work for a value of 1. In this case the loop in PRIMEP would become endless, since 2 is never a factor of 1 and so the exit condition, (FACTORP X N) would never be satisfied. This problem is avoided quite simply by beginning the search for prime numbers at 2. For completeness, the value of 1 is directly printed out. This also prevents anybody who has not seen the function explicitly stated from assuming an error has occurred. (PRINT 1) is, of course, indented one space for clarity of style.

After this, the DO loop begins. The loop variable X is initialised at the necessary value of 2 and incremented by 1 at each cycle. The exit condition at the beginning of the loop will prevent further execution as soon as X becomes equal to the number chosen by the user when PRIMES-TO is called N. The main body of the DO loop then consists of the conditional expression (IF (PRIMEP X) (PRINT X) T). This uses the new predicate function to print out only prime values of X. The three closing brackets mark the end of the IF condition, the DO loop and the function definition respectively.

PRIMES-TO is illustrated for numbers up to 50.

```
* (PRIMES-TO 50)
1
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
NIL
```

Discovering Perfect Numbers

It is important that, before any significant applications of LISP programming are attempted, complete familiarity is gained in the use of the control structures outlined in this chapter. In particular, a further example can be given of the use of the LISP conditional branching and iterative looping illustrated so far. This is another mathematical exercise in the field of number theory. It deals with a class of number probably less familiar than primes but equally intriguing. These are the 'perfect numbers'.

All composite numbers have factors, which can be added together to yield a new number. Thus 10 has, excluding itself, the factors 1, 2 and 5. These add to give 8, which is less than 10. Other numbers may have factors whose sum exceeds them. For example, the factors of 12 are 1, 2, 3, 4 and 6 and these add to 16. However, there is a scattering of numbers with factors which generate the number itself when they are added. The most easily discovered example of such a 'perfect' number is 6. Here the factors, 1, 2 and 3, do indeed add to 6.

The study of perfect numbers is of extremely long lineage. Euclid possessed a method for calculating even perfect numbers. Another early mathematician, Nicomachos of Gerasa, who lived in the first century A.D., is reputed to have discovered the next two perfect numbers after six, 28 and 496. Slightly later, Saint Augustine of Hippo (A.D. 354 - 430) wrote, 'Six is a number perfect in itself and not because God created all things in six days. Rather the inverse is true. God created all things in six days because this number is perfect.' The French mathematician and friend of Descartes, Marin Mersenne, also compiled a slightly erroneous list of perfect numbers, not corrected until 1947!

Attempting to locate perfect numbers simply by determining, and then adding, the factors of numbers selected blindly would be an impossible task without the number-crunching prowess of a computer. Fortunately, with a computer a very simple algorithm will suffice for testing whether a number is perfect. With a suitable program, plus, to be honest, a generous hint about the range in which to search, a piece of real numerical research can be carried out. An appropriate LISP function is developed here.

A Perfect Number Predicate Function - PERFECTP

`PERFECTP` is a predicate function. Given a perfect number as its argument, it will evaluate to `T`. Any other number will result in `NIL`. The name has been chosen to reflect the standard nomenclature for predicate functions.

All the algorithm requires is that a running total is kept, starting at 1, of the integers which divide into the number under test. Obviously, there is no point in continuing to check numbers which exceed half the test number, since they cannot possibly be factors. At this point, therefore, the sum is compared with the test number. If the two are equal, a perfect number has been discovered.

A flowchart for this process would appear as in Figure 2.2. This can then be coded as the predicate function, `PERFECTP`. The vital point is that it should return `T` for perfect numbers and `NIL` otherwise.

```
(DEFUN PERFECTP (N)
 (SETF TOT 0)
 (DOTIMES (X (INTEGER (/ N 2)) T)
          (IF (FACTORP (1+ X) N)
            (SETF TOT (+ TOT (1+ X))) T))
 (= TOT N))
```

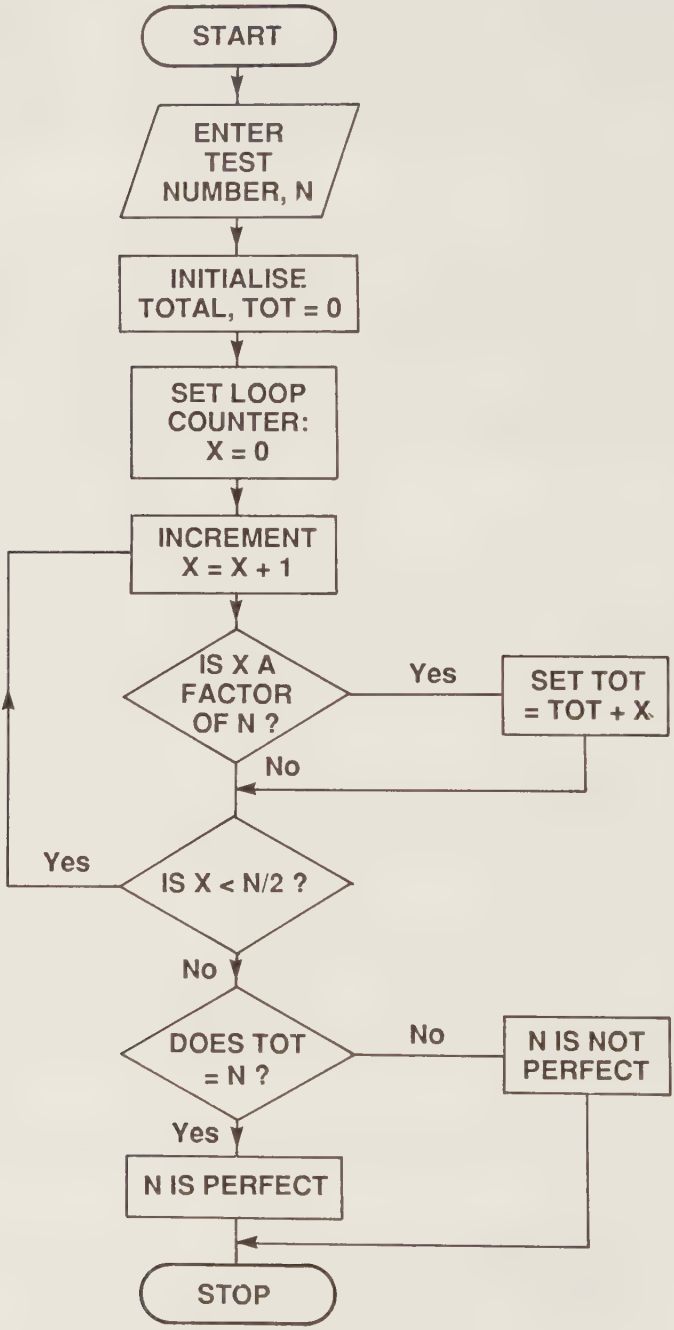

START
ENTER TEST NUMBER, N
INITIALISE TOTAL, TOT = 0
SET LOOP COUNTER: X = 0
INCREMENT X = X + 1
IS X A FACTOR OF N ?
Yes
SET TOT = TOT + X
No
Yes
IS X < N/2 ?
No
DOES TOT = N ?
No
N IS NOT PERFECT
Yes
N IS PERFECT
STOP

Figure 2.2

DEFUN shows that a function definition follows with the name PERFECTP. The parameter list simply contains the variable N, which is the number under test. This corresponds with the first instruction box of the flowchart. The second box, the initialisation of the total, is coded as (SETF TOT 0). The main loop of the program is then written as a DOTIMES construct. This is the obvious choice to make, because the loop has to repeat an exact number of times and these can be specified in advance. As stated above, there is no need to test any integers greater than half the value of the number being tested. This is (/ N 2). However, an integral value is required to control the DOTIMES loop and so (INTEGER (/ N 2)) is employed to define the number of loop cycles. The function INTEGER must be used rather than INT because no decimal component is permissible in governing a DOTIMES loop.

At the beginning of the first loop cycle the variable X has the value 0 rather than 1. Similarly, for each following cycle it has a value one less than that appropriate for the algorithm. To avoid this problem, and also the embarrassment of a 'division by zero' error from occurring in the first cycle, the variable is incremented before the conditional test is applied. This expression then has the form:

(IF (FACTORP (1+ X) N) (SETF TOT (+ TOT (1+ X))) T)

The condition simply uses the predicate written above to determine whether one number is a factor of another. If (1+ X) does divide into N, then it is a factor which must be added to TOT. The expression (FACTORP (1+ X) N) will evaluate to T in these circumstances and so the first action is triggered. Hence the total is adjusted by (SETF TOT (+ TOT (1+ X))), as required. If (1+ X) does not divide into N, the default second action T does not affect the total.

Conveniently, the last part of PERFECTP is able to employ a Golden Common LISP primitive to return the value of T or NIL needed. This is =, the mathematical predicate of equality. If TOT and N are equal the number is perfect. PERFECTP will need to evaluate as T. This is precisely the value returned by (= TOT N). Conversely, if the number is not perfect, the expression (= TOT N) will evaluate as NIL, which is again exactly what is wanted.

The new function can be tested with some of the perfect numbers already mentioned, as well as other numbers for a control. When checking the plausibility of any new function, a variety of examples should always be attempted.

```
* (PERFECTP 5)
NIL
* (PERFECTP 6)
T
```

The function can be seen to be working. As a final modification, the predicate function can now be inserted into a simple loop, which allows a series of numbers to be checked without the need to test each one individually.

The Search for Perfection - FIND-PERFECT

This function allows an upper and lower limit to be specified for the range of numbers which are being checked. The lower limit is represented by the variable A and the upper by B. These then govern the operation of a DO loop.

```
(DEFUN FIND-PERFECT (A B)
 (PRINT 'NUMBER-IS)
 (DO ((X A (1+ X)))
     ((= X (1+ B)))
     (IF (PERFECTP X) (PRINT X) T)))
```

The parameter list of the function has the two variables A and B. PRINT places 'NUMBER-IS' on the screen and then all the integers up to B are checked by the loop. In the exit condition, B is incremented by 1 to prevent premature departure from the loop before the value B itself is examined. The conditional expression then uses the newly-defined predicate function PERFECTP to determine whether any number within the range being checked is perfect. If such a number is found, it is added to the screen message. If no number is located, the only additional text on screen is the NIL, which appears in either case. The use of the function is illustrated below.

```
* (FIND-PERFECT 3 10)
NUMBER-IS
6
NIL
* (FIND-PERFECT 10 20)
NUMBER-IS
NIL
* (FIND-PERFECT 25 35)
NUMBER-IS
28
NIL
```

As implied above, the search for perfect numbers is unlikely to prove successful unless a lucky choice is made of the range being examined. Nevertheless, the function does demonstrate how LISP can be used in practical number theory, as well as illustrating further the application of loops and conditions.

Summary

1. Program control is achieved by a combination of looping, and branching.

2. A simple method of creating loops is to use DOTIMES.

3. DOTIMES creates an unconditional loop. It is therefore limited in its application.

4. The Boolean values of true and false are represented by the atoms `T` and `NIL` in LISP. These play a very significant role in LISP code.

5. A special category of LISP functions, predicates, tests their arguments to return either `T` or `NIL`.

6. `INTEGERP`, `>`, `<` and `=` are all predicate functions.

7. Predicates allow conditional branching to occur in LISP functions.

8. A general form of conditional is generated by `IF`.

9. A more generalised form is made available by `DO`. This includes an exit of a loop condition to permit conditional looping.

Exercise 2

1. Write a LISP function, `RAFFLE`, which prints either 'You have won' or `'Bad luck'` according to the identity of the user.

2. What values are printed by the loop
```
(DO ((X 7 (+ 4 X))) ((> X 30)) (PRINT X))?
```

3. Write a function which uses two nested `DOTIMES` loops to print out the first ten products of the multiples of 3 and 4, eg (3 X 4)=12, (6x8)=48...

4. Use `FACTORP`, and the fact that even numbers are divisible by 2, to write a new predicate function, `PARITYP`, which returns `T` for even numbers and `NIL` for odd. (The name avoids plagiarising an existing LISP predicate!)

3

Figuring It Out: Elementary Statistics

Introduction

Many people find it easier to learn something new if they are able to apply their knowledge fairly quickly to a practical problem. There are those, of course, who have no difficulty in holding large numbers of interrelating theoretical constructs indefinitely in their mind and who have little need for intellectual terra firma. But others may want to test themselves as they learn, by seeing how the concepts they are encountering fare when confronted by reality. LISP, especially, could appear excessively cerebral at first. It does not have the reassuring accessibility of that mainstay of beginners' programming, BASIC, with quasi-English vocabulary and transparent syntax. When approaching LISP for the first time it is sensible to attempt relevant applications of the language as soon as possible. Only in this way can essential confidence be gained.

The initial choice of the applications to be written as LISP code is rather critical. As suggested earlier, there is little point in selecting problems which themselves require a great deal of theoretical background before the necessary algorithms can be designed and written as LISP functions. The complexity of the task must not obscure the implicit objective of acquiring familiarity with LISP programming. To avoid this difficulty, it is perhaps sensible to attempt the first forays into LISP problem-solving in the domain of elementary statistics. Here, no demand for prior knowledge should exceed secondary school mathematics.

Although LISP has traditionally had a poor reputation in its handling of mathematical functions, it must be emphasised that this is not a criticism that can be fairly applied to Common LISP with its refined number-handling facilities. There is no difficulty of any kind in building up a useful library of statistical functions for the IBM PC-compatible, using the Golden Common LISP package. In typical LISP fashion, these functions will be based upon each another. One function may call another and, indirectly, further more fundamental functions. The subtle hierarchy established will lead to sophisticated functions of deceptive simplicity, yet capable of performing significant statistical analyses.

Recording Statistical Information in LISP

Practical statistics is, essentially, about handling sets of numbers. A student may be concerned about the implication of a series of test scores: 34, 30, 41, 23, 7. A health and safety representative might record the daily Fahrenheit temperature during January: 63.8, 64.0, 61.2, 62.6.

If a LISP statistics package is to be written to investigate series of numbers like these, an obvious way of entering the information into the program is to construct a list containing the numbers as its elements. The first series of numbers could be bound to a suitably named variable using SETF.

```
* (SETF SCORES '(34 30 41 23 7))
(34 30 41 23 7)
```

The information would always be available upon evaluating the variable.

```
* SCORES
(34 30 41 23 7)
```

Similarly, the variable TEMP could store the second set of numbers as a list.

```
* (SETF TEMP '(63.8 64.0 61.2 62.6))
(63.8 64.0 61.2 62.6)
```

Again the data could be displayed by typing the variable name:

```
* TEMP
(63.8 64.0 61.2 62.6)
```

Once numbers have been placed into lists like this, they can be analysed using typically list-oriented LISP functions. Nevertheless, it is essential that programs are written in a way which makes them easy for people to use. It would be extremely inconvenient to be obliged to construct a list with SETF each time that data needed to be entered into a program. Fortunately, it is easy to write a short function which uses READ inside a DO loop in order to build up a list from items typed in one by one at the keyboard. The name chosen for this new LISP function is INPUT. It requires, though, the introduction of three new LISP primitive functions first.

Common LISP Primitive - EQUAL

This is a predicate function, so it always returns a value of T or NIL. It takes two arguments and, if their structure is identical, EQUAL evaluates to T. Otherwise it returns NIL. Here is EQUAL testing whether two atoms are the same. Remember that they must be quoted to avoid an unsuccessful attempt at evaluation.

```
* (EQUAL 'PEA 'PEA)
T
* (EQUAL 'TWEEDLEDEE 'TWEEDLEDUM)
NIL
```

EQUAL can also be applied to lists.

```
    * (EQUAL '(HAROLD EDWARD JIM MAGGIE)
       '(HAROLD EDWARD JIM MAGGIE))
    T
    * (EQUAL '(CHARLES ANN ANDREW EDWARD)
       '(CHARLES ANN EDWARD ANDREW))
NIL
```

NIL was returned in the last example because, although the two lists contained the same elements, their order was different. As a result, the two lists do not possess identical structure.

Common LISP Primitive - APPEND

This primitive function can be used to create a new list by combining shorter ones.

```
* (APPEND '(A B C) '(D E F) '(G H I))
(A B C D E F G H I)
```

Common LISP Primitive - LIST

This function will return its arguments as a list.

```
* (LIST 'APPLE)
(APPLE)
* (LIST 'APPLE 'BANANA 'CHERRY 'DATE)
(APPLE BANANA CHERRY DATE)
```

Although they might appear very similar at first, it is essential to note an important distinction between the operation of LIST and APPEND. This can be illustrated by giving both functions the same arguments.

```
* (LIST '(W X) '(Y Z))
((W X) (Y Z))

* (APPEND '(W X) '(Y Z))
(W X Y Z)
```

The difference in the structure of the two lists obtained is obvious. Whether APPEND or LIST should be used in a particular case will depend on how the user intends the new list to be operated upon in later program code.

Building up a List - INPUT

With these two extra primitive functions available, INPUT can now be coded like this.

```
(DEFUN INPUT NIL
 (SETF LA NIL)
 (SETF EA NIL)
 (DO ((X 1 (1+ X)))
     ((EQUAL EA '(END)))
     (SETF LA (APPEND LA EA))
     (SETF EA (LIST (READ))))
 LA)
```

Here is how this function is constructed. First, DEFUN is used to begin the definition with the name INPUT. The parameter list is NIL because INPUT will not require any values to be passed to it when it is called. Everything it needs will be read from the keyboard. SETF is used twice to establish two new lists, LA and EA. Both of these will be NIL initially. LA is the list being constructed, element by element, from the user's keyboard entries. EA takes the value of each item typed in turn.

INPUT then uses a DO loop which will continue repeating until the condition ((EQUAL EA '(END))) becomes true. This means that until the user types END to indicate that the series of entries is complete, followed, of course, by pressing the return key or space bar, the function will add to the list LA. Because this exit condition is at the beginning of the DO loop, the entry END will not itself be added to the list being constructed.

Next in the loop is (SETF LA (APPEND LA EA)). Evaluation always begins at the most deeply embedded level. So first, (APPEND LA EA) produces a new list with the list EA added at the end of the current list LA. Then the SETF at the head of the main list updates LA to this new, lengthened form. This part of the function might seem a little disturbing if it was forgotten that the sublist was to be evaluated first.

The sublist (SETF EA (LIST (READ))) binds the keyboard entry to EA. LIST is used because EA needs to be a list when it is appended to LA. A closing parenthesis ends the DO loop and finally LA is left as the last item to be evaluated in INPUT. LA will therefore be the value of the whole function. The last value is always the one returned.

INPUT can now be used to create a list.

```
* (INPUT)
12.5
3
672
7
9.9
END

(12.5 3 672 7 9.9)
```

END has not appeared in the list. The function can be used to build up lists of non-numeric elements as well.

```
* (INPUT)
CALLISTO
GANYMEDE
EUROPA
IO
END

(CALLISTO GANYMEDE EUROPA IO)
```

With this function available to enter lists readily into other LISP functions, it is now possible to think about the first simple statistical programs that can be written.

Simple Statistics - the Mean, Median and Mode

Once a list of numbers has been entered by `INPUT` the problem of calculating the average value, already tackled above in a simpler way with the two functions `AVERAGE-A` and `AVERAGE-B`, could be approached again. The time a distinction will be drawn between the three types of 'average' which could be defined. The first of these is the arithmetic mean. This is the commonsense interpretation of an average. The arithmetic mean of a series of numbers is their total divided by the number of terms contributing to that total. The new LISP function which will calculate the mean is called `MEAN`.

The second type of average is the median. This is the middle value when all the numbers are arranged in order. The function written below for this average is `MEDIAN`.

The third type of average is the mode. This is the most frequently occurring value in a set of numbers. It is the practical type of average which would concern the buyer for a department store, more concerned with the number of people who have a size 16 collar rather than the arithmetic mean of all possible circumferences of neck. The function name, following the pattern now set, is `MODE`.

The first of these three programs, `MEAN`, will now be written. Before it can be developed, further primitive functions need to be introduced.

Common LISP Primitives - FIRST, SECOND, THIRD, NTH

When lists are being manipulated, it is essential that LISP can easily locate particular elements in them as required. It is especially important that the first element in the list can be obtained whenever needed. The function which does this is called `FIRST`, although the alternative name `CAR`, found in most other dialects of LISP, is also understood by Golden Common LISP. `FIRST` takes a list as its argument and returns its first element, as an atom.

```
* (FIRST '(MERCURY GEMINI APOLLO SHUTTLE)) MERCURY
```

The similarly named `SECOND` and `THIRD` behave as implied.

```
* (SECOND '(MERCURY GEMINI APOLLO SHUTTLE))
GEMINI

* (THIRD '(MERCURY GEMINI APOLLO SHUTTLE))
APOLLO
```

A generalised function, NTH, will obtain any element from a list. It has two arguments. The second is the list. The first is the position of the element in the list, assuming that counting begins from zero. Thus taking 0 as the first argument will simulate FIRST and taking 1 is equivalent to SECOND.

```
* (NTH 0 '(MERCURY GEMINI APOLLO SHUTTLE))
MERCURY

* (NTH 1 '(MERCURY GEMINI APOLLO SHUTTLE))
GEMINI
```

NTH is far wider in its application than this. It can operate on a series of numbers.

```
* (NTH 3 '(23 12 89 62 37))
62
```

NTH can be used with non-numeric lists or, of course, with variables representing lists.

```
* (SETF DRINK '(COKE WHISKEY GIN LAGER MILK))
(COKE WHISKEY GIN LAGER MILK)
* (NTH 0 DRINK)
COKE
* (NTH 4 DRINK)
MILK
```

With the function NTH, an algorithm can now be used which will find the mean of a series of numbers. This will be done after a final new function resident in Golden Common LISP has been illustrated.

Common LISP Primitive - LENGTH

The number of elements in any list typed into Common LISP can be found by applying the primitive function LENGTH.

```
* (LENGTH '(Pascal BASIC COBOL Ada LISP))
5
```

This is a convenient function for inclusion in new routines which the Common LISP user may devise. It is employed in the definition of MEAN.

The Average Value of a Series - MEAN

The earlier function developed, AVERAGE-B, calculated the average of a series of numbers entered via a DOTIMES loop. This meant that the total number of entries had to be specified in advance. If instead INPUT is used to enter numbers as a list, this restriction can be avoided. All that is then required is a function which will calculate the arithmetic mean of a list of numbers. Here is such a function, MEAN.

```
(DEFUN MEAN (L)
 (SETF TOT 0)
 (DOTIMES (X (LENGTH L) T)
          (SETF TOT (+ TOT (NTH X L))))
 (/ TOT (LENGTH L)))
```

The variable TOT is initialised by (SETF TOT 0) and then used in the DOTIMES loop to represent the running total of the list's elements. The number of times the loop repeats is controlled quite simply by using LENGTH in the sublist (X (LENGTH L) T). The function NTH will then return each of the elements in turn. They are added in turn on to the running total by the s-expression (SETF TOT (+ TOT (NTH X L))). The final bracket appearing after this list in the function marks the end of the DOTIMES loop. Once outside the loop, when all the elements have been added to the running total, LENGTH is used again to calculate the arithmetic mean.

Here is MEAN used to average a list:

```
* (MEAN '(2 9 6 4 3))
4.8
```

Again the power of LISP is illustrated by the way in which one user-defined function can readily summon another:

```
* (MEAN (INPUT))
7
2
11
5
END

6.25
```

It is in this form that MEAN becomes a generalised, and more convenient, alternative to the simpler function AVERAGE-B.

Top-down Programming, a Hierarchy of Functions

More significant than the mean in many cases is the middle value, or median. The next statistical function to be developed, MEDIAN, will find the median of a series. However, in

order to be able to do this, it will first need to be able to sort the elements of any list of numbers into order. After this, it will be evident which is the middle value. MEDIAN will use another function, ORDER, to perform the initial rearrangement of the list

There are a variety of ways in which a randomly arranged set of numbers can be sorted. ORDER employs a very simple method. It finds the largest number in the list and places this in a second list which it gradually builds up. The number is removed completely from the first list. The process is repeated until all the numbers have been shifted, in the correct sequence, from the first list to the second.

The algorithm used in ORDER also requires further functions as it operates. When it needs to find the largest number in the list it employs the user-defined function GREATEST. Similarly, each time the number has to be removed from the first list, ORDER calls REMOVE. This extracts a given element from a list, but if the element occurs more than once, only one example of it is erased. In contrast, Common LISP has the primitive DELETE which excises all examples of a selected element from a list. Using DELETE instead of REMOVE would create a drastically different list and not simply help to produce an ordered copy.

MEDIAN's use of ORDER, which in turn requires GREATEST and REMOVE, is yet another example of LISP's ability to build up a hierarchy of function calls. It also illustrates the way in which one problem, writing a function to calculate the median of a set of numbers, has led to a number of subsidiary problems. Each of these needs to be tackled before the overall task can be accomplished. The way in which a particular job must be resolved into its component parts before further progress is made is quite typical of programming in general. Correct programming, with any language, really requires the user to think carefully, and to jot down the main aspects of the problem on paper, before approaching the keyboard. With LISP it is even more important to adopt this approach. Success with the language depends upon the programmer's readiness to develop a series of fairly small functions which call one another. LISP demands a top-down attitude towards programming.

Before GREATEST, the first of the new functions needed by MEDIAN, can be written, another LISP primitive is needed.

Common LISP Primitive - REST

This function also has the name CDR. It takes a list as its argument and returns a new list which is identical, except that the first element has been removed.

```
* (REST '(A B C D E))
(B C D E)
```

It is important to realise that the original list still remains unchanged. This can be demonstrated by first binding a list to the identifier L.

```
* (SETF L '(EGGS SUGAR BUTTER FLOUR))
(EGGS SUGAR BUTTER FLOUR)
```

REST can now be given L as its argument.

```
* (REST L)
(SUGAR BUTTER FLOUR)
```

However, the unaltered list is still produced if L is evaluated.

```
* L
(EGGS SUGAR BUTTER FLOUR)
```

If it was necessary to alter L, SETF would be required.

```
* (SETF L (REST L))
(SUGAR BUTTER FLOUR)
```

L will now evaluate to the shorter list.

```
* L
(SUGAR BUTTER FLOUR)
```

GREATEST can now be written, for its subsequent use in ORDER and thus in MEDIAN.

Finding the Greatest Element - GREATEST

This is the definition of the function GREATEST.

```
(DEFUN GREATEST (L)
(SETF BIG 0)
(DOTIMES (X (LENGTH L) T)
         (IF (> (FIRST L) BIG)
          (SETF BIG (FIRST L)) T)
         (SETF L (REST L)))
 BIG)
```

First, SETF is used to initialise the variable BIG. At the end of the DOTIMES loop which follows, the list L is progressively decapitated by (SETF L (REST L)). This means that at each pass through the loop, BIG can be compared with all the elements of L simply by checking it against the first element of the ever shortening list. Also inside the loop is the conditional expression (IF (> (FIRST L) BIG) (SETF BIG (FIRST L)) T), which adjusts BIG to the largest value each time. Once the whole list is checked through, BIG will be equal to the greatest number. The final value given by the function is then just BIG, evaluated before the final closing parenthesis.

GREATEST is illustrated by these examples:

```
* (GREATEST '(14.7 2 19.3 5.5))
19.3

* (SETF A '(5 2 7 6 1))
(5 2 7 6 1)
* (GREATEST A)
7
```

Deleting a Single Element - REMOVE

The other new function required by ORDER and MEDIAN is REMOVE, which is coded like this.

```
(DEFUN REMOVE (E L)
 (SETF LB NIL)
 (DO ((X 1 (1+ X)))
     ((= E (FIRST L)))
     (SETF LB (APPEND LB (LIST (FIRST L))))
     (SETF L (REST L)))
 (APPEND LB (REST L)))
```

The function builds up a new list LB, which is identical to L, except that the first occurrence of the element E is deleted. First, LB is initialised as the empty list. In the DO loop, LB is then constructed, element by element, from the original list L. At each pass through the loop the first element of L is added to LB by (SETF LB (APPEND LB (LIST (FIRST L)))). The expression takes this form because (FIRST L) gives the initial element of L and then (LIST (FIRST L)) makes it into a list, as required by the APPEND function for its arguments. At the end of the loop the same element is then removed from L by (SETF L (REST L)). This means that on the following execution of the loop the next element of L will be added to LB. The process continues until the unwanted element E is detected by the exit condition incorporated into the loop ((= E (FIRST L))). The operation of the function then continues outside the loop and LB is joined to the rest of L, excluding the element E. Of course, if E occurs again in L it will escape any further deletion. This would not have occurred if DELETE had been used. As (APPEND LB (REST L)), the required list, is the last expression to be evaluated before the final closing bracket, it is the value returned by the whole function. The original list L remains intact outside REMOVE because it was present in the parameter list.

Here is REMOVE illustrated in operation:

```
* (REMOVE 6 '(3 6 2 9))
(3 2 9)
* (REMOVE 7 '(5 4 7 2 7 7))
(5 4 2 7 7)
```

One further LISP primitive function is needed before ORDER can be coded.

Common LISP Primitive - CONS

This function is, in fact, a very important LISP primitive. It allows the construction of a new list by adding elements, one by one, to the head of an existing list. Its general form is:

```
(CONS New head of list Existing list)
```

Here CONS is demonstrated.

```
* (CONS 'HUGHES '(HUNT MILLAIS ROSSETTI))
(HUGHES HUNT MILLAIS ROSSETTI)
```

It can be used, in conjunction with SETF, to build up a fresh list completely.

```
* (SETF L NIL)
NIL
* (SETF L (CONS 'C L))
(C)
* (SETF L (CONS 'B L))
(B C)
* (SETF L (CONS 'A L))
(A B C)
```

Now the function, ORDER, can be written.

Putting the Least First - ORDER

GREATEST and REMOVE are combined to give the required function ORDER.

```
(DEFUN ORDER (L)
 (SETF LC NIL)
 (DOTIMES (X (LENGTH L) T)
          (SETF LC (CONS (GREATEST L) LC))
          (SETF L (REMOVE (GREATEST L) L)))
LC)
```

The new version of the list LC, which will have all its elements arranged in ascending order, is initialised by (SETF LC NIL). Next, the DOTIMES loop uses (LENGTH L) to work through L just a sufficient number of times to consider each element of the list in turn. At each pass through the loop, (GREATEST L) yields the biggest number present. It is added by CONS to the new list being built up, LC. REMOVE then takes this greatest element out of L. At the next pass through the loop, the next biggest element is added to the front of LC and, again, is removed from L. As this process continues, all the numbers in L are reassembled into the ordered list LC.

ORDER operates on a list of numbers like this:

```
* (ORDER '(6 3 9 5 3))
(3 3 5 6 9)
```

It can also accept a list directly from the function INPUT.

```
* (ORDER (INPUT))
8
34
0
12
6
END

(0 6 8 12 34)
```

Now that ORDER has been defined, the function MEDIAN can be developed. One final LISP primitive must be demonstrated first.

Common LISP Primitive - EVENP

EVENP is another predicate function. It is a useful test of the parity of a number, returning T for even and NIL for odd numbers.

```
* (EVENP 6)
T

* (EVENP 7)
NIL
```

Locating a Middle Value - MEDIAN

When the median of a series is calculated, the numbers must first be placed in order. The value of the median will be calculated in one of two possible ways, depending upon whether the series is odd or even in length. If it is odd, the median is simply the middle value.

3, 6, 9, 12, 23, 45, 52 ... Median = 12

If there are an even number of items in the series, the median is calculated as the arithmetic mean of the middle two.

3, 6, 9, 12, 23, 45, 52, 88... Median = (12 + 23)/2 = 17.5

This distinction is incorporated into the algorithm used in MEDIAN. It is the reason why the primitive function EVENP required introduction. The function NTH is used again.

```
(DEFUN MEDIAN (L)
 (ORDER L)
 (SETF N (/ (LENGTH L) 2))
 (IF (EVENP (LENGTH L))
  (/ (+ (NTH N LC) (NTH (1- N) LC)) 2)
   (NTH (- N .5) LC)))
```

ORDER turns the list L into the sorted list LC. N is then made equal to half the length of the list. For the first example above N would be 3.5 and for the second N would be 4. The conditional structure uses EVENP to consider even series first. From the example given, it can be seen that for an even series the number placed Nth in the list must be averaged with the number preceding. This is calculation is performed by the expression (/ (+ (NTH N LC) (NTH (1- N) LC)) 2). An odd series will have N ending in 0.5 and the median will always be given by the preceding integral value. Thus, in the example, N = 3.5 and the median is the third number in the series. This is why the expression which calculates the median for an odd series is written as (NTH (- N .5) LC).

Here are examples showing how the function MEDIAN returns values for odd and even numbered lists.

```
* (MEDIAN '(5 2 1 7 4 3 2))
3

* (MEDIAN '(3 2 8 7))
5.0
```

Finding the Most Frequent Element - MODE

A third type of average is the mode. This is the most frequently occurring number in a series. Here is an example.

4, 1, 7, 4, 3, 2, 4, 2 ... Mode = 4

As an explicit exercise in top-down programming, the algorithm to calculate the mode of a list of numbers can be written in seven successive stages. These will all need to be incorporated into the final coding of the function MODE.

1st: Begin with the list of numbers: (4 1 7 4 3 2 4 2)

2nd: Sort the list into ascending order: (1 2 2 3 4 4 4 7)

3rd: Construct a second list, each element of which counts the frequency of the numbers occurring in the first: (1 2 1 3 1)

4th: Find the greatest element in this new list: 3

5th: Add together all the numbers up to this element: `1+2+1 = 4`

6th: Add `1: 4+1 = 5`

7th: Find the element in this position in the first list: `4`

The first two stages have their solutions already. The function `INPUT` will create the list of numbers and `ORDER` will sort it as required. The third stage does demand some fresh programming. The function used to build up the new list will be called `SUBTOTAL`. It, in turn, calls `NEW-ITEM`.

```
(DEFUN SUBTOTAL (L)
 (SETF LD NIL)
 (SETF EB 1)
 (DOTIMES (X (1- (LENGTH L)) T)
          (IF (= (NTH X L) (NTH (1+ X) L))
           (SETF EB (1+ EB))
            (NEW-ITEM)))
 (APPEND LD (LIST EB)))

(DEFUN NEW-ITEM NIL
 (SETF LD (APPEND LD (LIST EB)))
 (SETF EB 1))
```

The new list being constructed is initialised as `LD`. It is then built up from elements generated as the successive values of the variable `EB` in the `DO` loop. The loop needs to repeat just one time fewer than the length of the list, as determined by the value of `(1- (LENGTH L))`. This is because each element of `L` is being compared with the next and obviously the last element cannot be compared with any further element following it.

`SUBTOTAL` has to count the number of each group of identical elements as the loop progresses through `L`. This means that `EB` has to be incremented every time the next element encountered is equal to the current element. When a different element occurs, however, `EB` must be set back to 1. Additionally, the previous value of `EB` must then be placed into the list `LD`, as one of the elements representing an individual subtotal of L.

This algorithm is implemented by the conditional expression at the beginning of the loop. The test written into the conditional, `(= (NTH X L) (NTH (1+ X) L))`, checks whether two successive elements are equal. If the test returns `T`, the next part of the conditional is evaluated. `(SETF EB (1+ EB))` then simply increments the value of `EB`. If the test returns `NIL`, however, the function `NEW-ITEM` is called. It can be seen that this adds the value of `EB` as an element to the list `LD` and also resets `EB` to 1.

Finally, when the end of the list `L` is reached, the final value of `EB` must be added to `LD` as its last element. This is done by the expression `(APPEND LD (LIST EB))` after the `DO` loop. It is the value of this last expression which is returned by the whole function `SUBTOTAL`.

Fortunately, it does evaluate to the new list LD, as can be seen by these examples of SUBTOTAL applied to ordered lists of numbers.

```
* (SUBTOTAL '(1 1 1 2 2 3 4 4 4))
(3 2 1 3)

* (SUBTOTAL '(45 45 45 89 89))
(3 2)
```

The remaining four stages of the algorithm for MODE can now be attempted. They all code quite easily into the final version of the function.

```
(DEFUN MODE (L)
 (SUBTOTAL (ORDER L))
 (SETF TOT 0)
 (DO ((X 0 (1+ X)))
     ((= (NTH X LD)(GREATEST LD)))
     (SETF TOT (+ TOT (NTH X LD))))
 (NTH (+ 1 TOT) (ORDER L)))
```

The function SUBTOTAL needs to operate on a sorted list and so (SUBTOTAL (ORDER L)) is used to create the list LD. The DO loop then performs the fourth and fifth stages of the MODE algorithm described above. It works through LD, using NTH to find each element before it is added to the running total TOT by the expression (SETF TOT (+ TOT (NTH X LD))). When the greatest element is located, (= (NTH X LD) (GREATEST LD)) causes the loop to exit. The sixth and seventh stages of the algorithm then appear in the final expression (NTH (+ 1 TOT) (ORDER L)), which increments the running total and finds the appropriate element in the original list.

MODE returns the most frequently occuring element in a list like this.

```
* (MODE '(3 7 2 1 0 7))
7

* (MODE (INPUT))
8.9
3.5
2.2
3.5
3.5
8.9
4.7
END

3.5
```

Arrays and Storing Data

A familiar way of holding information in a computer program is to use an array. An array is a structure with a fixed number of locations, decided in advance, where data of an identical sort can be stored. Any particular item can be retrieved again by using its unique index to refer to the location where it is held. Arrays are therefore rather like a card index or a series of numbered boxes. They appear in other computer languages besides LISP.

Common LISP Primitives - MAKE-ARRAY, AREF

Golden Common LISP allows the user to set up arrays to hold information in a way which allows it to be retrieved easily. The function used to initialise an array is `MAKE-ARRAY`. It is followed by the number of array locations required to hold all the data. Shown here is an array initialised with five locations which will be used to name brands of whiskey. An appropriate name is chosen for the array, as all five whiskeys will be trans- Atlantic.

```
* (SETF BOURBON (MAKE-ARRAY 5))
```

When this is typed into Golden Common LISP a prompt will appear similar to this.

```
#<VECTOR T 5 72AB:024F>
```

Because it is always one dimensional, an array in Golden Common LISP is also a vector. The number after the `T` shows the number of array positions. The only other part of the prompt that will vary is the final two hex characters.

The array locations can be filled by use of the function `AREF`. This function name is a mnemonic for Array REFerence. When `AREF` is employed, the name of the array has to be stated as well as the particular location being addressed. Locations are numbered beginning with `0`. The five whiskeys may be inserted like this.

```
* (SETF (AREF BOURBON 0) 'OLD-CROW)
* (SETF (AREF BOURBON 1) 'JIM-BEAM)
* (SETF (AREF BOURBON 2) 'JACK-DANIELS)
* (SETF (AREF BOURBON 3) 'OLD-GRANDAD)
* (SETF (AREF BOURBON 4) 'WILD-TURKEY)
```

The array's data can be obtained directly.

```
* (AREF BOURBON 2)
JACK-DANIELS
```

The information in the array is accessed by the use of just one variable which gives each array location. The array elements are defined by a single subscript. This is because the array has only one dimension. Multi-dimensional arrays were not supported in the version of Golden Common LISP available when this text was written, but it is easy to arrange algorithms which circumvent this apparent restriction when programming in the language.

Putting Information into an Array - FILL-ARRAY

There is no difficulty in writing a short function which allows an array to be initialised and all its locations filled with data typed in directly.

```
(DEFUN FILL-ARRAY (N)
 (SETF AR (MAKE-ARRAY N))
 (DOTIMES (X N T)
          (SETF (AREF AR X) (READ))))
```

The function allows the user to decide how many locations are needed. If four pieces of information are to be held, the function will be called by `(FILL-ARRAY 4)`. This means that the parameter list will be `(N)`, where N is the number of array locations to be set up. `(SETF AR (MAKE-ARRAY N))` initialises an array with the name `AR` and the required number of locations. The `DOTIMES` loop then lets each location be filled with data from the keyboard by a suitable combination of `AREF` and `READ` in the expression `(SETF (AREF AR X) (READ))`. The two extra closing brackets, of course, relate to the `DOTIMES` loop and the initial `DEFUN`.

`FILL-ARRAY` is used like this.

```
* (FILL-ARRAY 5) AYER
RUSSELL
RYLE
STRAWSON
WITTGENSTEIN
T
```

The actual value returned by `FILL-ARRAY` is the `T` produced by LISP evaluating the `DOTIMES` loop. It would be possible to check that the function had worked correctly by directly interrogating particular locations in `AR`.

```
* (AREF AR 3)
STRAWSON
```

Clearly, though, `FILL-ARRAY`, needs a companion function which can readily display the content of the same array `AR` on the screen. This function, `SHOW-ARRAY`, will use a new LISP primitive, `PRIN1`, which will be illustrated first.

Common LISP Primitive - PRIN1

Each time the function `PRINT` is chosen to add information to the screen, it begins a fresh line of the display. There is another printing function, `PRIN1`, which does not do this. Instead, it places its information on a single line. Another useful feature in arranging a particular display is that `PRIN1` does not put a trailing space after the text it prints. This means that the programmer has a great deal of control over the precise formatting of the display being generated.

Here is an example of PRIN1 used inside a loop to produce a display occupying only one screen line.

```
* (DOTIMES (X 3 T)
    (PRIN1 'QUIET-PLEASE-))
QUIET-PLEASE-QUIET-PLEASE-QUIET-PLEASE
```

In a similar fashion, it is possible to create very compact yet legible screen displays when information needs to be presented to the user during the execution of a program. This will be illustrated in the function SHOW-ARRAY.

Displaying the Contents of an Array - SHOW-ARRAY

PRIN1 is used here in this function, which allows the contents of an array to be displayed on the screen.

```
(DEFUN SHOW-ARRAY (N)
 (DOTIMES (X N T)
          (PRIN1 (AREF AR X))
          (PRIN1 '_)))
```

N appears in the parameter list, because the number of items in the array must be known in order for them all to be printed out by the DOTIMES loop. The number will be the same as the one used initially when AR was set up by the function FILL-ARRAY. The loop then finds each item in the array in turn using the AREF function in the expression (AREF AR X). This is placed on the screen by (PRIN1 (AREF AR X)). Because PRIN1 has been used rather than PRINT, each item will be placed on the same line. To avoid confusion when successive items are printed without any separation between them, PRIN1 is also used to include the underline character within the loop. This spaces the display of the array's contents quite effectively.

SHOW-ARRAY may be used to print out the array contents entered by the earlier example.

```
* (SHOW-ARRAY 5)
AYER_RUSSELL_RYLE_STRAWSON_WITTGENSTEIN
```

So far none of the array examples given have involved numerical entries. When arrays are used to manipulate numbers, it is very easy to produce short functions which will again process simple statistical information. First, a function will be shown which calculates the arithmetic mean of numbers held in the locations of the AR array.

The Average Value in an Array - MEAN-ARRAY

MEAN-ARRAY assumes that AR has been filled with numerical data. It will return the arithmetic mean of all the elements in the array.

```
(DEFUN MEAN-ARRAY (N)
 (SETF TOT 0)
 (DOTIMES (X N T)
          (SETF TOT (+ TOT (AREF AR X))))
(/ TOT N))
```

Again the function needs to be given the number of elements in the array. The variable TOT is established by (SETF TOT 0) and used to hold the running total of the array elements generated by (SETF TOT (+ TOT (AREF AR X))) in the DOTIMES loop which follows. Outside the loop, the final expression (/ TOT N) means that the overall function will evaluate to the mean of the array elements.

MEAN-ARRAY can be used immediately after FILL-ARRAY to calculate the arithmetic mean of a sequence of numbers typed in on the keyboard.

```
* (FILL-ARRAY 5)6
4
2
1
7
T
* (MEAN-ARRAY 5)
4
```

The median of the elements of an array can also be calculated. As with the earlier example involving the median of the elements of a list, it is necessary first to be able to arrange the array elements in ascending order. This is possible with the next function.

A Bubble Sort in LISP - SORT-ARRAY

The algorithm used in this function is quite different from the one employed in sorting list elements. There a completely new list was assembled by removing the largest element each time from a successively depleted list. Now that an array is being used, however, it is extremely easy to identify particular elements by their designated array location. As a result of this, sorting can take place by the bubble-sort algorithm in which elements are exchanged between different locations according to their magnitude.

This rule simply states that array elements should be considered in pairs and their order reversed if the the larger element occurs first. This process is repeated until eventually no more swapping is required. At that point all the elements are in an ascending sequence.

The operation of the algorithm requires an extra variable, here called SF for Sorting Flag, which indicates by its value of 1 or 0 whether the array elements are in order, as required. A value of SF = 1 corresponds to the array elements not being in ascending order, whereas SF = 0 indicates that the array is properly sorted.

```
(DEFUN SORT-ARRAY (N)
 (SETF SF 1)
 (DO ((X 1 (1+ X)))
      ((= SF 0))
      (SETF SF 0)
      (DOTIMES (Y (1- N) T)
               (IF (> (AREF AR Y)
                (AREF AR (1+ Y)))
                 (SWAP)))))
```

The flag is initialised by `(SETF SF 1)`. Two loops follow in the function. The outer loop's variable is `X`, but this is not actively involved in the algorithm. The `X` loop simply repeats until the exit condition `(= SF 0)`, is met by the variable `SF`. The investigation of the current order of the numbers in the array, and their resorting, is performed by the inner loop.

The way in which the sorting algorithm operates is understood most easily if the value of the `SF` variable is followed during the execution of the function. At first it will be `1`. This avoids the unbound variable error which would occur when the condition `(= SF 0)` is encountered. Immediately after this, however, `SF` is made equal to `0`. This is because the algorithm's operation assumes that the numbers are in the correct order already and that the loop should exit on the next pass, unless it is detected that they are not in the ascending order. The inner `DOTIMES` loop then determines whether or not the elements are ordered. It does this by moving through the array locations, comparing `(AREF AR Y)` with the following location, `(AREF AR (1+ Y))`. Note that the range of this loop needs to be one less than the total number of array elements. The `IF` condition will call the embedded function `SWAP` if two elements are incorrectly ordered.

```
(DEFUN SWAP NIL
 (SETF AE (AREF AR Y))
 (SETF (AREF AR Y) (AREF AR (1+ Y)))
 (SETF (AREF AR (1+ Y)) AE)
 (SETF SF 1))
```

This moves the two array elements into their correct positions, using the variable `AE` like the empty box which might be used to hold odd items temporarily while the spare room was being cleared out. Equally importantly, `SWAP` changes the value of `SF` from `0` to `1`. This acknowledges the fact that the array was not in ascending order and results in another check being made by the `Y` loop. Eventually, though, all the array elements will be correctly sorted. A complete pass through the `Y` loop will then occur with no calls to `SWAP`. The value of `SF` will therefore remain as `0` and the condition in the main loop, `(= SF 0)`, will at last be met. The `X` loop then exits and the function ends, with the array sorted.

The operation of the algorithm being used can be demonstrated very effectively by inserting the earlier function `SHOW-ARRAY` at the end of the `SWAP` function, just before the final closing bracket, together with a brief message to explaining that sorting is taking place. This will mean that every time the numbers require rearrangement their current order will be

displayed. It is much easier, then, to appreciate precisely how the routine works. SWAP will now look like this.

```
(DEFUN SWAP NIL
 (SETF AE (AREF AR Y))
 (SETF (AREF AR Y) (AREF AR (1+ Y)))
 (SETF (AREF AR (1+ Y)) AE)
 (SETF SF 1)
 (PRINT 'SORTING)
 (SHOW-ARRAY N))
```

Here is a typical example with this modification to SWAP.

```
* (FILL-ARRAY 6)7
4
8
2
1
0

T
* (SORT-ARRAY 6)SORTING
4_7_8_2_1_0_SORTING
4_7_2_8_1_0_SORTING
4_7_2_1_8_0_SORTING
4_7_2_1_0_8_SORTING
4_2_7_1_0_8_SORTING
4_2_1_7_0_8_SORTING
4_2_1_0_7_8_SORTING
2_4_1_0_7_8_SORTING
2_1_4_0_7_8_SORTING
2_1_0_4_7_8_SORTING
1_2_0_4_7_8_SORTING
1_0_2_4_7_8_SORTING
0_1_2_4_7_8_SORTING
NIL
```

The new function can be applied in determining the median value of numbers held in an array.

The Median Value in an Array - MEDIAN-ARRAY

MEDIAN-ARRAY analyses an array of numbers and returns the median, as shown on the next page:

```
(DEFUN MEDIAN-ARRAY (N)
 (SORT-ARRAY N)
 (SETF M (/ N 2))
 (IF (EVENP N)
  (/ (+ (AREF AR (STRIP-ZERO M))
   (AREF AR (STRIP-ZERO (1- M)))) 2)
    (AREF AR (STRIP-ZERO (- M .5)))))
```

Variable `M` is equated to half of `N` after the function has used `SORT-ARRAY` to place the numbers in ascending order. What is now returned depends upon whether there are an even or odd number of array elements altogether. This is tested by `(EVENP N)`. If the number is even, the first action of the `IF` conditional is evaluated. This calculates the average of the two middle elements. From the example it can be seen that these are the `Mth` and `(M-1)th` locations.

Even number of array elements: N = 6 M = 3

```
Array location:  0 1 2  3  4  5
Number in array: 2 5 9 11 15 17
```

Middle elements = Locations 2, 3 = Location (M - 1), M

The expression adding these two middle elements needs to use `STRIP-ZERO`, so that the array locations can be found:

```
(/ (+ (AREF AR (STRIP-ZERO M)) (AREF AR (STRIP-ZERO (1- M)))) 2)
```

If the number is odd, the second action of the `IF` conditional is evaluated. From the second example it follows that the median is the element in location `(M - 0.5)`.

Odd number of array elements: N = 7 M = 3.5

```
Array location:  0 1 2  3  4  5  6

Number in array: 2 5 9 11 15 17 23
```

Middle element = Location 3 = Location (M - 0.5)

The LISP expression for this is `(AREF AR (STRIP-ZERO (- M .5)))`.

The function now operates like this:

```
* (FILL-ARRAY 7)5
23
11
15
```

```
2
9
17

T
* (MEDIAN-ARRAY 7)
11
```

Scientific Notation in Golden Common LISP

A letter `D` is used to signify scientific notation in Golden Common LISP and separates th significant digits from the exponent part.

$2.0D0 = 2.0 \times 10^0 = 2$

$1.4D3 = 1.4 \times 10^3 = 1400$

$5.3D\text{-}4 = 5.3 \times 1/10^4 = 0.00053$

Newton-Raphson Calculation of a Square Root

In many statistical investigations, it is necessary that the square root of certain quantities i obtained. Golden Common LISP does contain a square root function, SQRT, but, like man mathematics functions, this requires that the machine running LISP has the Intel 808 numeric processor extension. As many lower priced IBM PC-compatible personal computer which have entered the market, do not possess this facility, a square root function has to b written using the LISP primitives available. Fortunately, this is not difficult.

There is a standard iterative formula which can be applied to the solution of an equation. Thi is the Newton-Raphson algorithm, which assumes that if X1 is an initial approximate solutio for the equation $f(X) = 0$, then a closer approximation, X2, is given by a simple relationship.

$$X2 = X1 - f(X1) / f'(X1)$$

The first derivative of f(X) is represented by $f'(X)$. This expression for the approximatio has to be applied to the equation linking a number, N, with its square root, X.

$$N = X^2$$

In the notation being used, this gives for f(X) and f'(X) these two expressions:

$$f(X) = X^2 - N$$

$$f'(X) = 2X$$

Inserting these into the equation above yields the following.

$$X2 = X1 - (X1^2 - N)/2X1$$

$$X2 = X1 - X1/2 + N/2X1$$

$$X2 = X1/2 + N/2X1$$

$$X2 = (X1 + N/X1)/2$$

This equation giving a better approximation for a square root converges very rapidly. Here, it is used to find the square root of 9, taking half of 9 as the initial approximation. Within the accuracy of the ordinary pocket calculator used, an exact answer has been given without a single zero in the display after only four iterative stages.

Table 3.1

First approximation - X1	*Second approximation - X2*
4.5	3.25
3.25	3.009615
3.009615	3.000015
3.000015	3

This method for finding a square root can be employed in a function, SQROOT. First, another Common LISP primitive must be introduced.

Common LISP Primitive - ABS

The function ABS returns the absolute value of a number, equal to the number when it is positive and to the number multiplied by minus one when it is not.

```
* (ABS 5)
5

* (ABS -5)
5
```

The ability to obtain an absolute value is frequently essential in algorithm design.

A Square Root Function in LISP - SQROOT

It is evident that the algorithm used above is quite powerful. Conveniently, the expression, X2 = (X1 + N/X1)/2, deriving a closer approximation, X2, from a less accurate one, X1, can be coded into LISP directly as the function APPROX.

```
(DEFUN APPROX (N R)
 (/ (+ R (/ N R)) 2.0D0))
```

The parameter list contains a variable, N, which is the number for which the square root i being calculated, and a second variable, R, representing the current approximation. Th function uses the Newton-Raphson algorithm to obtain a better approximation. The LIS code is equivalent to the equation deduced above. LISP's scientific notation has been used fo the divisor. The 2 of the equation has been replaced by 2.0D0 in the LISP expression. Thi is necessary for the exit condition to be met in the overall function being developed SQROOT.

Clearly, the function APPROX is not sufficient in itself to calculate a square root. It must b used iteratively inside another function, so that it can be repeatedly called until th approximate value for the square root which it returns is deemed sufficiently accurate. A companion function is helpful in checking upon this accuracy. The function is called CLOSE.

```
(DEFUN CLOSE NIL
 (< (ABS (- N (* R R))) 1.0D-11))
```

This works as follows. If R is a good approximation for the square root of N, then there wil be little difference between N and R multiplied by itself. As N may be either larger or smalle than this product, the absolute value must be taken, leading to (ABS (- N (* R R))) a the expression showing how close R is to being an accurate square root of N. CLOSE the compares the difference with the arbitrarily small quantity 0.00000000001. The overal function acts as a predicate. It returns T if R is a good approximation and NIL otherwise. I can therefore be used as the exit condition in the overall square root function SQROOT.

```
(DEFUN SQROOT (N)
 (SETF R (/ N 2.0D0))
 (DO ((X 1 (+ 1 X)))
     ((CLOSE))
     (SETF R (APPROX N R)))
 R)
```

The initial approximation for the square root is taken as simply half of N. This was the firs value assumed in the example worked through earlier for the square root of 9. The LIS expression which assigns the first approximation to R is therefore coded as (SETF R (/ N 2.0D0)). After this follows a DO loop in which the X variable simply repeats the loop rather than taking any active role in the loop's contents. Within the loop successively close approximations for the square root are obtained by feeding the current value back into th function APPROX. This is carried out by (SETF R (APPROX N R)). The loop will exi when the condition (CLOSE) yields T for an acceptable value of R. This is then returned a the evaluation of the last R before the final closing parenthesis.

The function is demonstrated here.

```
* (SQROOT 81)
9.0
```

Measuring the Dispersion of Numbers

The square root function which has now been written can be employed in a new function, which calculates the standard deviation of a set of numbers held in an array.

The standard deviation is an important measure of the spread, or dispersion, of data. Two sets of numbers might possess the same arithmetic mean, median and mode and yet exhibit a totally different degree of clustering.

```
<---------SPREAD OF DATA---------> MEAN MEDIAN MODE

18, 19, 20, 20, 24, 25, 27, 31, 32  24    24    20

 1,  4, 20, 20, 24, 26, 30, 39, 52  24    24    20
```

Here, each series of nine numbers has a total of 216, giving an arithmetic mean of 216/9 = 24. The fifth, or middle, value for each is 24 and is therefore the median. In each set, the value 20 occurs twice, making this the mode. The two sets, nevertheless, have quite a different spread of values between the greatest and least.

There are a number of ways of indicating just how much sets of numbers like these are spread out. The most obvious, giving the range between the highest and lowest numbers, is unsatisfactory because extreme values at either end of the set might imply a wider spread than genuinely appropriate to the remaining numbers. Similarly, the interquartile range, between the items one-quarter and three-quarters of the way through the range of numbers, is not employed as a reasonable measure of the spread, as too much of the data is omitted.

The best indication of spread is found from the deviation of each number from the mean, XM. These deviations are first squared to avoid the problem of negative values. The arithmetic mean of the squared deviations is found and then the square root of this quantity calculated, giving the standard deviation. The equation for this measure of spread is therefore:

$$S = [((X1-XM)^2 + (X2-XM)^2 + (X3-XM)^2 + ...(XN-XM)^2)/N]^{1/2}$$

It can be applied to both sets of numbers given above.

First set of data:

Value	18	19	20	20	24	25	27	31	32
Deviation	-6	-5	-4	-4	0	1	3	7	8
Deviation^2	36	25	16	16	0	1	9	49	64

Mean of squared deviations = (36+25+16+16+0+1+9+49+64) / 9 = 24

Standard deviation = $(24)^{1/2}$ = 4.9

Second set of data:

Value	1	4	20	20	24	26	30	39	52
Deviation	-23	-20	-4	-4	0	2	6	15	28
Deviation^2	529	400	16	16	0	4	36	225	784

Mean of square deviations = (529+400+16+16+0+4+36+225+784) / 9 = 223.33
Standard deviation = $(223.33)^{1/2}$ = 14.94

The standard deviation has clearly distinguished the two. A LISP function can be written to perform precisely the same calculation.

A LISP Function to Measure Dispersion - DEVIATION

Here is how the final new function in this chapter is defined.

```
(DEFUN DEVIATION (N)
 (SETF M (MEAN-ARRAY N))
 (SETF SUM 0)
 (DOTIMES (X N T)
          (SETF SUM (+ SUM (SQUARE
          (- (AREF AR X) M)))))
(SQROOT (/ SUM N)))
```

The expression `(SETF M (MEAN-ARRAY N))` binds the arithmetic mean of the numbers in the `AR` array to the variable `M`. In the `DOTIMES` loop this value of `M` is then used to calculate the deviation for each of the numbers in turn as `(- (AREF AR X) M)`. The `SQUARE` function from Chapter One is applied to each deviation and then the running total of the squared deviations obtained with the overall expression `(SETF SUM (+ SUM (SQUARE (- (AREF AR X) M))))`. Outside the loop, `SUM` is divided by `N` to give the arithmetic mean of the squared deviations. Finally, the new `SQROOT` function operates to return the standard deviation as the value of `DEVIATION`.

`DEVIATION` can be applied to the first set of data like this.

```
* (FILL-ARRAY 9)18
19
20
20
24
25
27
31
32

T
* (DEVIATION 9)
4.89897948556636D0
```

Summary

1. Number-crunching programs can be written easily in LISP to perform simple statistics.

2. EQUAL is a predicate function to test s-expressions, returning T if their structure is the same.

3. APPEND combines lists into a single list.

4. LIST returns all its arguments as a new list.

5. APPEND and LIST are quite distinct in their operation.

6. FIRST returns the first element in a list, SECOND selects the second and THIRD the third.

7. Any arbitrary element can be selected from a list by using the more general function, NTH.

8. LENGTH returns the number of elements in a list.

9. REST returns a list without its head.

10. CONS adds a new element at the head of a list.

11. EVENP is a predicate which detects even numbers.

12. Arrays can be set up and accessed using MAKE-ARRAY and AREF.

13. PRIN1 is an alternative output function.

14. ABS returns the absolute value of a number.

Exercise 3. Evaluate the following:

1. `(APPEND '(A B) '(C D) '(E F))`

2. `(LIST '(A B) '(C D) '(E F))`

3. `(FIRST '(ALAN BOB CHRIS DAVID))`

4. `(SECOND '(A (B C) D E))`

5. `(NTH 3 (REVERSE '(A B C D E F G H)))`

6. Write a function which uses an array to keep a record of phone calls and their duration.

7. Create SIDE, which gives the length of the remaining side of a right-angled triangle, given two sides either of which may be the hypotenuse.

4

Minding Your Own Business... Time Series and Input-Output Analysis

Forecasting and Time Series

Enough LISP has been now been outlined to allow the development of a set of functions which investigate time series. Time series analysis is an important tool in business planning. It involves recording numerical information, often sales figures, over a reasonably long period of time and then attempting to identify underlying patterns revealed by the data. If such patterns can be found, they help in extrapolating future projections.

Care has to be taken in such analysis. If past figures are to be a guide towards the future, it has to be assumed that a moderately steady, equilibrium situation exists. Numbers obtained by means of a mathematical model cannot be regarded as a substitute for the human expert. Nevertheless, interpreted with caution and with past experience of the practical details involved, the figures generated by a computer program can be of immense help in guiding decision making.

One model which is frequently used in time series analysis assumes that the variation observed in recorded data is the result of different components added together. First, there is the long term trend. For example, the number of passengers passing through a certain sea port may gradually increase year by year. Secondly there is a cyclical variation superimposed on to the trend. To continue with the same example, certain foreign resorts with which the port is linked may undergo a fluctuating popularity amongst tourists. A third component is the seasonal variation. Probably fewer passengers will be travelling by sea in the winter. Finally, there is a random variation which will further modify the figures. If an important sporting event occurs, many more people may decide to visit the country. The overall relationship can be symbolised by this equation.

Variation = Long-term Trend + Cyclical Variation + Seasonal Variation + Random Variation

A simple list of the passengers carried by the port's ferry companies will be the result of all these factors. Combined in an apparently complicated fashion, they will obscure

the underlying trend and make planning for the companies very difficult. Fortunately, the numerical analysis of such information into component parts, using the methods of time series, is precisely the type of number handling for which a computer is ideally suited.

Analysising a Time Series

The way in which data can be analysed into its component variations is best explained with a numerical example. In the table below, the quarterly sales figures (in £ million!) of an imaginary firm are recorded over a four year period. It will be seen that the figures have been analysed into columns showing trend, seasonal and random deviation. This has been achieved by a method involving the smoothing out process of moving averages.

Table 4.1

Year	Quart	Quart Figs	Four-point	Centred Trend	Dev.	Seas. dev.	Rand. dev.
1	1	28	-	-	-	-	-
	2	34		-	-	-	-
			32				
	3	36		31	5	-0.67	5.67
			30				
	4	30		29	1	2	-1
			28				
2	1	20		30	-10	4.67	-14.67
			32				
	2	26		33	-7	-6	-1
			34				
	3	52		40	12	-0.67	12.67
			46				
	4	38		48	-10	2	-12
			50				
3	1	68		48	20	4.67	15.33
			46				
	2	42		50	-8	-6	-2
			54				
	3	36		53	-17	-0.67	-16.33
			52				
	4	70		53	17	2	15
			54				
4	1	60		54	6	4.67	1.33
			54				
	2	50		51	-1	-6	5
			48				
	3	36		-	-	-	-
	4	46	-	-	-	-	-

The entries of the table have been obtained by using a pocket calculator. The first and second columns simply show the period of four years covered by the time series and the third gives the recorded quarterly figures during this time. The fourth column gives

the four-point moving average. This means that the quarterly figures have been added in groups of four and the mean taken for each. The first four-point average is (28+34+36+30)/4 = 32. The second four-point average is (34+36+30+20)/4 = 30. This calculation is repeated until all the quarterly figures have been analysed. These averages naturally represent a figure for the middle of the relevant period and have to be placed on lines between the rows of the table.

The processed information is placed back in step with the quarterly figures by calculating the centred trend. This is the arithmetic mean of the column of four-point average, now taken in pairs. The first centred trend will be (32+30)/2 = 31 and the second centred trend is (30+28)/2 = 29. All the values for the centred trend are recorded in the fifth column.

Once the centred trend has been calculated, the deviation shown by the quarterly figure from this value can be found immediately by subtracting the trend from it. The first row of the table for which this can be done represents the third quarter for year one. Here, the deviation is (36-31) = 5. The second deviation, entered just as all the others are in the sixth column, is (30-29) = 1. The third deviation, (20-30) = -10, shows that the deviations can be negative as well as positive. The deviation from the centred trend will be assumed here to be just a combination of two superimposed influences, the seasonal variation and the random variation. The time series analysis now has to attempt to separate out the seasonal deviation and random deviation caused by these.

The seasonal deviation is found by first calculating the average value of the deviation for each quarter over all the years involved. There is no value for the first quarterly deviation for the first year, but the values for the second, third and fourth years are -10,20,6. The arithmetic mean is (-10+20+6)/3 = 5.33. Similarly, the mean deviation calculated for the second quarter is (-7-8-1)/3 = -5.33. For the third quarter the mean deviation is (5+12-17)/3 = 0. Finally, the fourth mean deviation can be seen to be (1-10+17)/3 = 2.67. The total deviation for all four quarters must be zero, but instead here it is 2.67. It can be reduced to zero by modifying each quarterly deviation with the adjustment (-2.67)/4 = -0.67. This generates the four seasonal deviations 4.67,-6,-0.67,2 shown in the seventh column.

The random deviations recorded in the eighth column are found by subtracting the seasonal deviation from the actual deviation in the sixth column.

When the entire table has been calculated it can be used in predicting future sales and in general long-term planning. The table itself may be produced via a series of short LISP functions, the first of which is explained after the introduction of two primitives resident in Common LISP.

Common LISP Primitive - FORMAT T

`FORMAT T` is a function which allows further refinement of the screen display and is used as an alternative to the function `PRINT`. It is a convenient way of placing

messages on the screen in a conventional and readable fashion, rather than as the lists of words which have been displayed each time PRINT has been employed. The argument given to FORMAT T is the required text, placed between a pair of double quotes. These do not appear when the expression is evaluated. This test function will illustrate its operation.

```
(DEFUN TESTA NIL
 (FORMAT T "A boat, beneath a summer sky"))

* (TESTA) A boat, beneath a sunny sky
NIL
```

The expression evaluates to NIL, but this does not usually distract the user's attention because in most practical circumstances the word NIL will not appear on the screen. This would only occur if FORMAT T were to be used at the very end of a function which was not then being followed by some further evaluation. A worse problem is clearly the position on the screen where the text has appeared. It is solved by the next function.

Common LISP Primitive - TERPRI

TERPRI can be used to improve a screen display still further by forcing a fresh line to be adopted. It is like a carriage return and, in fact, the name given to the function comes from TERminal PRInt. It is shown operating here in TESTB.

```
(DEFUN TESTB NIL
 (TERPRI)
 (FORMAT T "A boat, beneath a summer sky")
 (TERPRI)
 (FORMAT T "Lingering onward dreamily")
 (TERPRI)
 (FORMAT T "In an evening of July -"))

* (TESTB)
A boat, beneath a summer sky
Lingering onward dreamily
In an evening of July -
NIL
```

FORMAT T and TERPRI are both used in DATA, the first of the functions which need to be defined in order to carry out time series analyses.

Entering Quarterly Figures - DATA

The DATA function allows quarterly information for a time series to be entered:

```
(DEFUN DATA NIL
 (TERPRI)
 (FORMAT T "How many years?")
 (TERPRI)
 (SETF NA (READ))
 (SETF NB (* NA 4))
 (FORMAT T "Type quarters separately.")
 (TERPRI)
 (FILL-ARRAY NB))
```

DATA has NIL for its parameter list because no local variables are involved in its operation. It begins with the function TERPRI placing a subsequent screen prompt on the line beneath the entry that the user has typed, (DATA). This looks much neater and is easier to follow. FORMAT T then requests the period over which the time series is to be calculated. Again TERPRI causes the display to move to a fresh screen line. The number of years selected is bound to the variable NA by the assigning expression (SETF NA (READ)). NA is later required by the functions QUART-DEV and RANDOM-DEV. The number of yearly quarters equivalent to this value of NA is calculated as the variable NB and used for a call to the earlier function FILL-ARRAY. This function is used after a final FORMAT T, which gives further information to the user about how to enter the quarterly figures. TERPRI is employed a third time before FILL-ARRAY allows the information entered to be placed in the suitably dimensioned array AR.

Four-point Moving Average - FOUR-POINT

The second column of the table is calculated by the function FOUR-POINT.

```
(DEFUN FOUR-POINT NIL
 (SETF NC (- NB 3))
 (SETF FOUR-PT (MAKE-ARRAY NC))
 (DOTIMES (X NC T)
          (SETF TOT 0)
          (DOTIMES (Y 4 T)
                   (SETF TOT (+ TOT (AREF AR (+ X
                    Y)))))
          (SETF (AREF FOUR-PT X) (/ TOT 4))))
```

It can be seen that the number of places in the column is three fewer than in the first. This is an inevitable consequence of calculating a four-point moving average. NC is assigned the necessary value by (SETF NC (- NB 3)) and used to initialise a fresh array to hold the moving averages in the further expression (SETF FOUR-PT (MAKE-ARRAY NC)). The values to be inserted in the array are derived from the quarterly figures held in AR by the calculation performed inside the two nested DOTIMES loops. In these, the value of the loop variable X moves through all the AR locations, while the Y value provides the increments needed to identify the four array

elements contributing to each average. For example, when X = 0 the Y values 0, 1, 2, 3 operating in the expression (AREF AR (+ X Y)) select the quarterly figures 28,34,36,30 from the first column. These are added inside both loops to give the running total TOT. Outside the Y loop, the value of TOT is used to place the relevant four-point moving average in the array FOUR-PT. The calculation of the average and assignment to the array is done by (SETF (AREF FOUR-PT X)(/ TOT 4)). In the case of the values quoted, this will give an average of (28+34+36+30)/4 = 32, which is placed in the first location of the array FOUR-PT. More generally, the range of the X DOTIMES loop ensures that all the four-point moving averages are generated and placed in their array.

Two-point Moving Average - CENTRED-TREND

The function CENTRED-TREND can now be used to process the values in the FOUR-PT array and calculate the two-point averages for the third column.

```
(DEFUN CENTRED-TREND NIL
 (SETF ND (- NB 4))
 (SETF TREND (MAKE-ARRAY ND))
 (DOTIMES (X ND T)
          (SETF (AREF TREND X)
           (/ (+ (AREF FOUR-PT X)
          (AREF FOUR-PT (1+ X))) 2))))
```

The number of places in the third column is now four less than the original number of items entered and so an appropriate value is derived for the variable ND from NB by (SETF ND (- NB 4)). The array for the centred trend values is TREND. This is set up, using the value of ND, by (SETF TREND (MAKE-ARRAY ND)). After this, the values for TREND are calculated in the DOTIMES loop. Just one loop is required, in comparison with the more complicated algorithm used in FOUR-POINT, because only two values need to be averaged to obtain the two-point average. The two numbers to be added from the FOUR-PT array for the Xth value in the TREND array are obviously (AREF FOUR-PT X) and the number which follows it, (AREF FOUR-PT (1+ X)). The sum of these is divided by 2 and assigned by SETF to the relevant location in the new array (AREF TREND X).

Deviation from the Trend - TREND-DEV

The sixth column of the table records the deviation from the trend. Each entry here is the result of subtracting the centred trend from the original quarterly information in the same row. These values are produced by the function TREND-DEV.

```
(DEFUN TREND-DEV NIL
 (SETF DEV (MAKE-ARRAY ND))
 (DOTIMES (X ND T)
          (SETF (AREF DEV X)(- (AREF AR (+ 2 X))
          (AREF TREND X)))))
```

There will be a deviation calculated to correspond with each value of the centred trend. This means that the total number of entries in the fourth column will also be ND. The array for these figures DEV is therefore initialised to hold the new values by (SETF DEV (MAKE-ARRAY ND)). Because the centred trend is two places out of step with the original quarterly entries, the location in the AR array which corresponds to the Xth position in the TREND array will be (+ 2 X). This in turn means that the Xth deviation is (- (AREF AR (+ 2 X))(AREF TREND X)). It is this value that is assigned to the Xth location of the DEV array in the DOTIMES loop.

Mean Deviation for Each Quarter - QUART-DEV

Before the seasonal deviations can be calculated for the seventh column, the mean deviation for each of the four quarters must be found by considering the relevant values for the deviation over the entire period concerned. QUART-DEV is the function which uses the information in the DEV array to calculate the mean deviation for each quarter.

```
(DEFUN QUART-DEV (QR)
 (SETF ST (IF (> 2 QR)(+ QR 2)(- QR 2)))
 (SETF TOT 0)
 (DOTIMES (X (1- NA) T)
          (SETF TOT (+ TOT (AREF DEV (+ ST
           (* X 4))))))
 (/ TOT (1- NA)))
```

A problem is caused by the fact that the values in the DEV array move sequentially through all the quarters, beginning with the third quarter of the first year. In order to calculate the mean deviation for any particular quarter over the total number of years covered by the time series, the figures in the DEV array need to be rearranged in order to match them with each quarter. Examining the table of values shows that the array locations in DEV relate to the quarters as shown below. Note that the quarterly values are each one less than the 'expected' number. This is to match the starting value in the subsequent DOTIMES loop.

QUARTER	ARRAY LOCATIONS IN DEV			
0	-	2	6	10
1	-	3	7	11
2	0	4	8	-
3	1	5	9	-

The deviations in locations 2, 6 and 10 give the mean deviation for the first quarter. Locations 3, 7 and 11 give the second quarter. Similarly, the third quarter's mean deviation is calculated from DEV locations 0, 4 and 8 and the fourth quarter's from locations 1, 5 and 9.

An arithmetical relationship has to be found between the number representing the quarter QR and the array locations needed in DEV. It can be seen that the locations increment regularly by 4 and so all that is necessary is to derive an expression which links the quarterly numbers, QR = 0, 1, 2, 3, and the initial array positions, 2, 3, 0, 1. A clear pattern exists. If QR is not less than 2, the array location is found by subtracting 2 from QR's value. Thus QR = 2 gives location 0 and QR = 3 leads to location 1. Conversely, if QR is less than 2 the location is found by adding 2 to QR. Then, QR = 0 gives location 2 and similarly QR = 1 identifies location 3. This simple algorithm is incorporated in the expression for ST, the 'starting variable' in QUART-DEV, `(SETF ST (IF (> 2 QR) (+ QR 2) (- QR 2)))`. This illustrates the convenient way LISP allows the evaluation of a conditional expression to be assigned to a variable. It is a valid construction because of the inherent priority of evaluation, the most embedded sublists being the first to attract LISP's attention.

Once this essential value for ST is obtained, QUART-DEV simply uses a DOTIMES loop to calculate the average of the quarterly variations. The number of times the loop has to be executed is one less than the total number of years over which the time series applies. This is calculated by `(1- NA)`, which is also used to divide the running total outside the loop. All the necessary DEV locations are addressed by the arithmetical expression employed `(+ ST (* X 4))`. QUART-DEV does not itself place any information directly into an array. The final part of the function to be evaluated, `(/ TOT (- NA 1))`, is simply the mean deviation for the quarter selected and it is this value which is returned by the function.

Seasonal Deviation - SEASONAL-DEV

This function calculates the four values for the seasonal deviation which are repeated throughout the seventh column.

```
(DEFUN SEASONAL-DEV NIL
 (SETF SUM 0)
 (SETF SEASONAL (MAKE-ARRAY 4))
 (DOTIMES (X 4 T)
          (SETF SUM (+ SUM (QUART-DEV X))))
 (SETF ADJ (* (/ SUM 4) -1))
 (DOTIMES (X 4 T)
          (SETF (AREF SEASONAL X)
           (+ (QUART-DEV X) ADJ))))
```

First, the variable SUM is initialised and the array SEASONAL established to hold the four values for the seasonal deviation. In the first DOTIMES loop which follows, QUART-DEV is used to add together all four quarterly deviations. The total is held as SUM. The average is then found by dividing by four. The adjustment needed for the seasonal deviations, ADJ, is obtained by multiplying this average by a further -1, giving the overall expression `(SETF ADJ (* (/ SUM 4) -1))`. After this, the second DOTIMES loop uses `(+ (QUART-DEV X) ADJ)` to place modified quarterly deviations into the array SEASONAL.

Random Deviation - RANDOM-DEV

The figures for the final column of the table are calculated by the function RANDOM-DEV. This subtracts the adjusted seasonal deviation from the original deviation in the sixth column to show the random deviation.

```
(DEFUN RANDOM-DEV NIL
 (SETF RANDOM (MAKE-ARRAY ND))
 (DOTIMES (X (1- NA) T)
          (DOTIMES (Y 4 T)
                   (SETF NE (+ (* X 4) Y))
                   (SETF NF (IF (> 2 Y)(+ Y 2)
                    (- Y 2)))
                   (SETF (AREF RANDOM NE)
                    (- (AREF DEV NE)
                   (AREF SEASONAL NF))))))
```

First, an array, RANDOM, is initialised to hold the random deviations after they have been calculated. A problem then occurs in matching the array locations in the DEV array with those in the SEASONAL array. The former increment regularly by 1, beginning at 0, whilst the latter repeatedly assume the same four values, 2,3,0,1.

To resolve this, two nested DOTIMES loops are employed, together with two further variables, NE and NF, both calculated from the values assumed by the DOTIMES variables X and Y.

The maximum value of X, (1- NA), is one less than the number of years entered. In the case of the original example, this will be 3. X therefore has the values 0,1,2 as the outer DOTIMES loop executes. Y takes the values 0,1,2,3. The values taken by the third variable, NE, are (4X+Y) in standard, non-LISP notation, and can be seen to be the following.

Table 4.2

X	Y	N = 4X+Y
0	0	4x0+0 = 0
0	1	4x0+1 = 1
0	2	4x0+2 = 2
0	3	4x0+3 = 3
1	0	4x1+0 = 4
1	1	4x1+1 = 5
1	2	4x1+2 = 6
1	3	4x1+3 = 7
2	0	4x2+0 = 8
2	1	4x2+1 = 9
2	2	4x2+2 = 10
2	3	4x2+3 = 11

The range of the nested loops, combined with the expression chosen for NE, has

generated the required locations for the DEV array. All of these locations could have been obtained far more directly with one DOTIMES loop, but by using the separately nested loop for Y, numbers are now also available which determine the locations in the SEASONAL array. Although Y begins in the order 0,1,2,3, and the array locations are in the order 2,3,0,1, the adjustment is made quite easily by the expression used to define NF, (SETF NF (IF (> 2 Y)(+ Y 2)(- Y 2))). This converts the sequence 0,1,2,3 into the rearranged series 2,3,0,1.

At the end of the function, the values acquired by NE and NF are used to perform the correct subtractions and to insert the new values into the RANDOM array.

Screen Display

All the mathematical functions have now been written which allow the calculation of a time series. The final detail is to decide upon a convenient way in which the information can be presented to the user. Screen display is always important and becomes a special problem when using limited graphics facilities. To supplement the language, Golden Common LISP can support a specialised graphics environment, called HALO, developed by Media Cybernetics. This is a software package written in assembler and capable of interfacing with a wide variety of IBM PC-compatible machines running under MS-DOS. HALO can be combined with different high-level languages as well as LISP, including BASIC, FORTRAN, C and Pascal. It provides a wide range of additional commands for the LISP programmer, with primitives which allow the construction of points, lines, arcs, circles and ellipses. Various types of fill are available.

An investigation of the possibilities of HALO, unfortunately, is beyond the scope of the current text. Nevertheless, with a little care and imagination, it is possible to produce quite effective displays with only a limited set of commands. It will be shown below how a function, DISPLAY, can be written which approximates very closely to the typical layout conventionally used in the display of the results of a time series calculation. The techniques described here will also be suitable for application in other areas. Before DISPLAY is written, two other functions, LINE and PAUSE, will be illustrated which may also be useful in various program displays. Both new functions will then be incorporated into DISPLAY. Initially, as an introduction to LINE, another Common LISP primitive, FLATC, will be explained.

Common LISP Primitive - FLATC

In designing a display, it is frequently important to know exactly how much of a screen line will have to be devoted to a certain piece of information. The neatly ordered rows and columns of a table, for example, would not be possible if an unexpectedly long entry forced the remaining data towards the right-hand side of the monitor. Fortunately, there is a Common LISP function called FLATC which returns the number of characters in an item and this can be used to solve the problem, as shown below.

```
* (FLATC 'STEPHEN-HAWKING)
15

* (FLATC 18.247)
6
```

This function can be put to immediate use in writing the further function, LINE.

Improving a Screen Display - LINE

The neat columns needed in the later function DISPLAY are produced with the help of LINE. This uses the underline character to produce an extended line the length of which can be related to the size of a given item already on the screen. By making this an inverse relationship, so that the line becomes shorter as the number of characters in the item increases, it is possible to keep constant the total screen width occupied by both the item and its associated line. This results in automatic formatting of columns of data, provided that the total width of each column has been chosen so that it will always be bigger than any possible item it needs to contain. This requires a little thought, first, about the possible range of the data that is being displayed. The width chosen here for the columns in the time series display is 15. The function is written, therefore, like this.

```
(DEFUN LINE (L)
 (DOTIMES (X ( - 15 (FLATC L)) T)
          (PRIN1'_)))
```

The parameter list contains L. This is the item for which the line is being drawn. The DOTIMES loop then simply displays the underline character a number of times related to the number of characters in L. This, of course, is given by the LISP primitive, FLATC. If L has just one character, then the loop will repeat 14 times. If L has 13 characters, then the underline character will be printed just twice. Any value of L will lead to a line which just completes the column to the correct width. PRIN1 is used, naturally, to make sure that all the underline characters appear in succession on the same line of the display. Wider or shorter columns are possible if the value of the number, 15, is increased or decreased.

The line linking the columns has a further advantage, because it makes it easier for the eye to travel along a row of the display and recognise which of the items in the various columns are associated with one another.

Slowing it Down - PAUSE

Computers process their information extremely quickly and can display results very rapidly. It is certainly the case that Golden Common LISP will be able to present data on the monitor at a rate which exceeds the average user's ability to assimilate it. Two

solutions for this communication bottle-neck are available when designing a program. A 'page mode' can be adopted for the display, in which only one screen of information is presented at a time. Here, the user is requested to press a key before a fresh display is shown. Alternatively, the program can be deliberately made to run more slowly when the screen needs to be studied in detail. New information is added at a rate which allows an acceptable scrolling of the display, adjusted to the speed with which the average person will be able to read everything shown. This approach is adopted here with the function PAUSE.
PAUSE simply involves two nested DOTIMES loops which automatically generate a brief delay while they are executed.

```
(DEFUN PAUSE NIL
 (DOTIMES (X 10 T)
          (DOTIMES (Y 10000 T))))
```

The length of the delay required will depend upon the actual material being displayed, its context within the overall program and other relevant factors. The particular machine upon which the LISP package is being run will also affect the time taken to execute the loops. When it has been decided how long the display should pause, the range of the variables can be selected. It is likely that, as here, two nested loops will be needed.

Printing out the Results - HEADING, INDENT, DISPLAY

DISPLAY will show five columns of a time series calculation, the four-point moving average, the centred trend, the deviation, the seasonal deviation and the random deviation. For clarity, each column needs to be named. A separate function HEADING, is provided to do this.

```
(DEFUN HEADING NIL
 (TERPRI)
 (PRIN1 'FOUR-POINT-AV__)
 (PRIN1 'CENTRED-TREND__)
 (PRIN1 'DEVIATION_____)
 (PRIN1 'SEASONAL-DEV___)
 (PRIN1 'RANDOM-DEV_____)
 (TERPRI))
```

TERPRI is first used to begun the table of values on a fresh line. PRIN1 is then employed five times to add each of the titles in turn to the same line of the display. The titles are linked by the underline character in a way which creates the columns of 15 character widths, as chosen above. After this, TERPRI again places the next displayed item on a new line.

When the table is printed on the screen, the rows containing the four-point averages will need to be offset from the others. This means that a line 15 characters long will be

required to occupy the equivalent position in the other rows, beneath the four-point average. INDENT uses a DOTIMES loop and the underline character to create this line.

```
(DEFUN INDENT NIL
 (TERPRI)
 (DOTIMES (X 15 T)
          (PRIN1 '_)))
```

The last four functions developed, LINE, PAUSE, HEADING and INDENT, all now contribute to DISPLAY.

```
(DEFUN DISPLAY NIL
 (HEADING)
 (DOTIMES (X (1- NA) T)
          (DOTIMES (Y 4 T)
              (SETF N (+ (* X 4) Y))
              (SETF S (IF (> 2 Y)(+ Y 2)
               (- Y 2)))
              (PRINT (AREF FOUR-PT N))
              (PAUSE)
              (INDENT)
             (PRIN1 (AREF TREND N)) (LINE
              (AREF TREND N))
             (PRIN1 (AREF DEV N)) (LINE
              (AREF DEV N))
             (PRIN1 (AREF SEASONAL S)) (LINE
              (AREF SEASONAL S))
             (PRIN1 (AREF RANDOM N)) (LINE
              (AREF RANDOM N))))
 (PRINT (AREF FOUR-PT (1- NC))))
```

First, HEADING places the column titles on the display. Two nested DOTIMES loops are then used to generate the array locations needed to identify the relevant items of data. This added complication arises from the fact that, although the entries in the other columns are each determined by a simple increment of their array variable, the seasonal variation in the array SEASONAL needs to be located by a variable which repeats the same four numbers in a cycle, rather than assuming steadily ascending values.

Fortunately, precisely the same problem was solved for the function RANDOM-DEV and now two variables, N and S, can be calculated in the same way from the values assumed by the DOTIMES variables X and Y. The items in the four-point average, centred trend, deviation and random deviation columns are all selected by N. The entries in the seasonal deviation column are determined, instead, by the variable S.

The first column of the table is displayed on the screen by (PRINT (AREF FOUR-PT N)). PAUSE is then called to give a short delay in the rate at which the display scrolls. Because the four-point average should not appear on the same line as the rest of

the array information, INDENT is used to force a new line and to fill the space in the column beneath the four-point average. After this the information in the TREND, DEV, SEASONAL and RANDOM arrays are placed on the screen. The use of PRIN1 means that the associated values appear on the same line of the display. After each item, LINE is called, using the previous array item as its argument, to place the next item in the correct column on the screen.
At the very end of DISPLAY, and outside the two loops, the last four-point average is added to the table by the final expression (PRINT (AREF FOUR-PT (1- NC))).

Now that the display function has been written, the overall time series program can be assembled from all its component routines. As it will also involve two more functions resident in Common LISP, these must be illustrated first.

Common LISP Primitive - NULL

NULL is a predicate function. It always returns either T or NIL as its value. If the list to which it applies evaluates to be NIL or the empty list, NULL returns T. Otherwise it returns NIL. It simply reverses the value of the list and the LISP programmer soon becomes familiar with its behaviour.

```
* (NULL (= 4 5))
T
```

Here, 4 does not equal 5 and so the embedded list evaluates as NIL. NULL operating on this value gives T.

```
* (SETF VULCAN 'MR-SPOCK)
MR-SPOCK
* (NULL (EQUAL 'MR-SPOCK VULCAN))
NIL
```

Now NULL works the other way. VULCAN does evaluate to MR-SPOCK and so (EQUAL 'MR-SPOCK VULCAN) will evaluate as T. Hence the overall list gives NIL.

Common LISP Primitive - Y-OR-N-P

In making a program user-friendly, it is often useful to include a prompt on the screen which outlines a choice available in its execution. The user is then asked to make an appropriate decision and type 'Yes' or 'No' in response to the choice offered. This becomes such a frequent feature that the software author frequently writes a suitable prompt routine to add to a personal tool-kit for inclusion in future programs.

Common LISP makes this unnecessary by offering a primitive function which will deal with such a situation. It is the predicate Y-OR-N-P. It can be used to present text on

the screen, offering a choice of action. This automatically is followed by a Y or N prompt. If the Y key is pressed, the word 'Yes' appears on the screen and the function returns T.

```
* (Y-OR-N-P "Is there more data?")
Is there more data? (Y or N) Yes
T
```

If N is chosen instead, 'No' is displayed and the value returned by the function is NIL.

```
* (Y-OR-N-P "Is there more data?")
Is there more data? (Y or N) No
NIL
```

The value of T or NIL returned by the predicate is immediately available in determining the resultant course of the program involved.

Calculating a Time Series - TIME-SERIES

The complete time series function can finally be written. It provides another example in which the top-down approach that has been adopted is clearly visible in the overall program structure.

```
(DEFUN TIME-SERIES NIL
 (DO ((X 1 (1+ X)))
     ((NULL (Y-OR-N-P "Type time series?")))
     (DATA)
     (FOUR-POINT)
     (CENTRED-TREND)
     (TREND-DEV)
     (SEASONAL-DEV)
     (RANDOM-DEV)
     (DISPLAY)))
```

All the functions which have been developed to contribute towards the calculation of a time-series are enclosed within a DO loop. The exit condition at the beginning of the loop uses the Y-OR-N-P predicate to allow repeated examples of different series to be investigated until the user wishes to do something else. The display will begin with the prompt 'Type time series?' If N is pressed, the Y-OR-N-P function will return NIL. Because the overall expression involves NULL, however, the exit condition itself evaluates as T. The exit condition is thus met, the loop is not executed and nothing further happens. Alternatively, if the user types Y, the Y-OR-N-P function yields T and the exit condition will have the value NIL. This means that the loop does not exit and so all the functions enclosed in it are called in turn, which allows data to be typed in and the complete time series to be displayed on the screen.

and so all the functions enclosed in it are called in turn, which allows data to be typed in and the complete time series to be displayed on the screen.

Here, for example, is TIME-SERIES applied to the original example.

```
* (TIME-SERIES)
Type time series? (Y or N) Yes
How many years?
4
Type quarters separately.
28
34
36
30
20
26
52
38
68
42
36
70
60
50
36
46

FOUR-POINT AV    CENTRED-TREND    DEV.    SEASONAL-DEV    RANDOM-DEV

32.0
                 31.0             5.0     -6.66667F-01    5.66667
30.0
                 29.0             1.0      2.0            -1.0
28.0
                 30.0            -10.0     4.66667        -14.6667
32.0
                 33.0             -7.0    -6.0            -1.0
34.0
                 40.0             12.0    -6.66667F-01    12.6667
46.0
                 48.0            -10.0     2.0            -12.0
50.0
                 48.0             20.0     4.66667         15.3333
46.0
                 50.0             -8.0    -6.0            -2.0
```

```
_________53.0________17.0_______2.0_____________15.0______
54.0
_________54.0________6.0_______4.66667__________1.33333__
54.0
_________51.0________-1.0______-6.0_____________5.0______
48.0
Type time series? (Y or N)
```

LISP and Input-Output Analysis

Another area in which LISP on an IBM personal computer can help the small business is in developing software which will apply the theory of input-output analysis to the organisation of resources. This is the method for investigating the way in which different sectors of a firm mutually interact with one another, as well as studying their relationship with external factors.

Vital to the study of input-output analysis is the distinction between those associated entities which can be regarded as part of the system under consideration and the environmental factors which are external to the system and yet with which it is inevitably involved. This division of a given situation into separate components on opposite sides of a system boundary will be seen later on to be an essential concept in computer modelling and in simulation as a whole.

The schematic relationship between system and environment can be visualised as a diagram as in Figure 4.1.

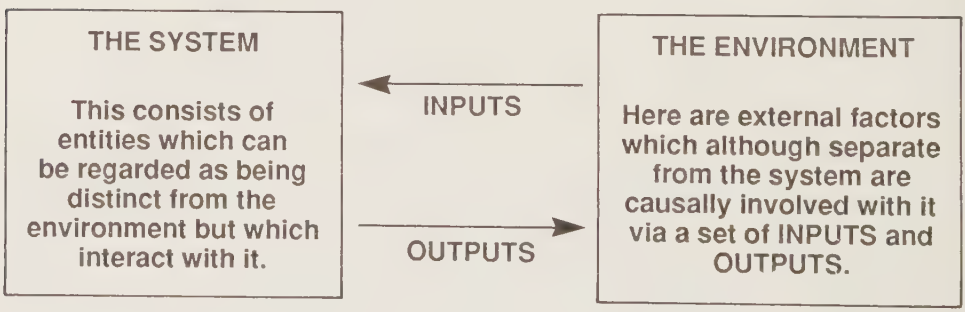


Figure 4.1

A problem is immediately encountered because the different parts of the system and of

the environment might have incommensurate inputs and outputs. To give a trivial example, two 'entities' within the system might be two typists, working in a (non-electronic) office, who loan each other correcting fluid, paper clips and carbon paper. They might well achieve a perfectly happy equilibrium involving a system of barter. Nevertheless, the office will probably be supported by a central stores unit which insists upon all stock orders being made on a particular size, and suitably numbered, pink requisition slip. Attempting a simple mathematical model which is able to equate these very different units for inputs and outputs will be extremely difficult. Quite evidently, some form of standardised units of exchange are going to be essential. In the real world the same principle applies.

In order to relate the inputs and outputs across the system boundary, as well as internal exchanges, all quantities are converted, where possible, into monetary sums. In this way, the arithmetic is vastly simplified. In the examples given below, all numbers will therefore refer to quantities of money, measured in bland 'units'.

The basic principles of input-output analysis can be studied by considering a firm which consists of three reasonably distinct sub-units or sectors. These are the entities within the system as described above. These three sectors provide inputs, and accept outputs, from one another. Naturally, the firm does not exist in magnificent isolation, splendidly contemplating the corporate navel. Although input-output analysis could, in the last analysis, be used to investigate the grander examples of bureaucracy, and idly quantify the rustling of paper, inputs and outputs to and from the environment are also included in the example.

The three units of the firm will be called sector X, sector Y and sector Z. For convenience these are shortened to SX, SY and SZ. The total output of SX is 60 units. Of these, 30 units are input back to SX itself, 16 units are inputs to SY, 8 units are inputs to SZ and 6 units are outputs from the system as a whole to meet external requirements. All of this information is recorded on the first row of Table 4.3.

Table 4.3

	Input to SX	Input to SY	Input to SZ	External	Total Output
Output from SX	30	16	8	6	60
Output from SY	12	12	16	8	48
Output from SZ	12	16	4	0	32
Balancing inputs	6	4	4	-	-
Total Input	60	48	32	-	-

The second and third rows indicate the distribution of outputs from the sectors SY and

SZ. Again for each of these the outputs will be directed to each of the three sectors and the external requirements. The final column gives the total output for SX, SY and SZ.

The information in these first three rows can then be used to complete the table by adding a bottom row to indicate the total input to each sector. For example, SX has an input of 30 units from itself, 12 units from SY and 12 units from SZ. These add to 54 units.

Immediately a discrepancy can be seen. The total output from SX is 60 units, as recorded in the final column, and yet the total input appears to be only 54 units. A balancing input of 6 units must be added so that the total input matches the total output for SX. These six units will represent inputs from outside the system, as well as being partly the result of the profit being made by the firm. The important factor operating here is that the total input to any sector must also be equal to the total output from it. This is one of the basic assumptions of input-output analysis.

Table 4.3 can be converted into a simpler mathematical structure. If the figures for SX are considered, it can be seen that 30/60 of SX's input comes from SX, 12/60 comes from SY and 12/60 comes from SZ. These fractions are known as the technical co-efficients. Together with the six fractions corresponding to the data for sectors SY and SZ, they form the input-output matrix:

```
l 30/60     16/48      8/32 l
l                           l
l 12/60     12/48     16/32 l
l                           l
l 12/60     16/48      4/32 l
```

To match the way matrices will later be displayed on the screen, the opening and closing brackets for this matrix are represented as columns formed by the lower case letter L. The matrix is simplified to:

```
l 1/2 1/3 1/4   l
l               l
l 1/5 1/4 1/2   l
l               l
l 1/5 1/3 1/8   l
```

This matrix can now be used to calculate what the new outputs from SX, SY and SZ should become if the external requirements alter. The assumption is made that even if these do change the internal operation of the system will not be affected so much that the proportions of SX's input coming from SX, SY and SZ are different. Similarly, the proportions of SY's and SZ's inputs obtained from all three sectors are assumed to be unaltered. A second basic principle of input-output analysis is that the co-efficients in the matrix remain the same, despite modifications to the output. This is an assumption which can be challenged if large changes in the outputs occur and the model would

then need to be adjusted.

Suppose the new external requirements for the three sectors are EX, EY and EZ and that the adjusted total outputs become X, Y and Z. Then for sector SX:

Total output, X = EX + SX's output to SX + SX's output to SY + SX's output to SZ

From the original table it follows that:

SX's output to SX = 30/60 x SX's total input

SX's output to SY = 16/48 x SY's total input

SX's output to SZ = 8/32 x SZ's total input

At this point the vital substitution is made that, for all sectors, total input is equal to total output. This gives:

SX's output to SX = 1/2 x X

SX's output to SY = 1/3 x Y

SX's output to SZ = 1/4 x Z

Entering these terms into the original equation now gives:

$$X = EX + 1/2\ X + 1/3\ Y + 1/4\ Z$$

These two additional equations are similarly derived:

$$Y = EY + 1/5\ X + 1/4\ Y + 1/2\ Z$$

$$Z = EZ + 1/5\ X + 1/3\ Y + 1/8\ Z$$

The three equations are most easily expressed in matrix form. At this stage the input-output matrix reappears. Using the normal method of multiplying matrices together, working along the rows of the input-output matrix and down the single column of the matrix representing the three total outputs, the final three terms of the equations become the matrix product:

```
1  1/2  1/3  1/4  1  1  X  1
1                 1  1     1
1  1/5  1/4  1/2  1  1  Y  1
1                 1  1     1
1  1/5  1/3  1/8  1  1  Z  1
```

This in turn leads to the matrix form of the equations:

```
1 X 1     1 EX 1     1 1/2  1/3  1/4 1  1 X 1
1   1     1    1     1                1  1   1
1 Y 1  =  1 EY 1  +  1 1/5  1/4  1/2 1  1 Y 1
1   1     1    1     1                1  1   1
1 Z 1     1 EZ 1     1 1/5  1/3  1/8 1  1 Z 1
```

The equations are simplified by first remembering that a matrix remains unchanged when multiplied by the unit matrix.

```
1 1 0 0 1   1 X 1      1 EX 1    1 1/2  1/3  1/4 1  1 X 1
1         1   1   1    1    1    1                1  1   1
1 0 1 0 1   1 Y 1  =   1 EY 1 + 1 1/5  1/4  1/2 1  1 Y 1
1         1   1   1    1    1    1                1  1   1
1 0 0 1 1   1 Z 1      1 EZ 1    1 1/5  1/3  1/8 1  1 Z 1
```

A rearrangement gives:

```
1 1 - 1/2  0 - 1/3  0 - 1/4 1   1 X 1      1 EX 1
1                           1   1   1      1    1
1 0 - 1/5  1 - 1/4  0 - 1/2 1   1 Y 1  =   1 EY 1
1                           1   1   1      1    1
1 0 - 1/5  0 - 1/3  1 - 1/8 1   1 Z 1      1 EZ 1
```

And finally:

```
1 -1/2  -1/3  -1/4 1   1 X 1      1 EX 1
1                  1   1   1      1    1
1 -1/5   3/4  -1/2 1   1 Y 1  =   1 EY 1
1                  1   1   1      1    1
1 -1/5  -1/3   7/8 1   1 Z 1      1 EZ 1
```

The problem of finding the three new values for the outputs has been reduced to obtaining the solution of three linear equations. The LISP functions which will handle such a situation are now developed. To aid explanation, this particular numerical example will no longer be used. Instead a set of equations with small, integral coefficients is given as an illustration.

The Solution of Equations in LISP - SOLVE-EQN

In fact, there are a variety of situations in which linear equations occur. It would be convenient if new LISP functions were available which provided immediate values for the variables in a set of equations like these:

$$3x + 2y + z = 5$$
$$x - y + 3z = 6$$
$$2x + y + z = 4$$

The necessary code will now be developed to solve a system of equations with three variables. The standard method of solution, involving the calculation of the inverse matrix of the coefficients of the variables, will be used and the principles could be extended to systems of higher rank. A number of functions will be required. `INPUT-MATRIX` stores the co-efficients of the variables as a 3 x 3 matrix. `DETERMINANT` calculates the associated determinant. `INVERSE-MATRIX` provides the new matrix required for the solution of the equations and `PRINT-INVERSE` displays this matrix if required. Finally, `SOLVE-EQN` finds the values of the three variables which satisfy the equations and displays the results clearly.

Entering a Matrix in LISP - INPUT-MATRIX

The first function which will be needed by the overall program `SOLVE-EQN` is `INPUT-MATRIX`. This allows the user to type in the elements of a 3 x 3 matrix. It can be employed, for example, in storing a matrix of the form:

```
1      1      1

2     -3      1

2     -2      1
```

This matrix will be the one used in some of the examples that follow. Note that it is not the matrix involved in the solution of the equations above.

The function is coded as follows.

```
(DEFUN INPUT-MATRIX NIL
 (SETF LC (MAKE-ARRAY 3))
 (SETF MC (MAKE-ARRAY 3))
 (SETF RC (MAKE-ARRAY 3))
 (DOTIMES (X 3 T)
          (TERPRI)
          (FORMAT T "Row No.")
          (PRIN1 (1+ X))
          (TERPRI)
          (SETF (AREF LC X) (READ))
          (SETF (AREF MC X) (READ))
          (SETF (AREF RC X) (READ)))
 (TERPRI)
 (FORMAT T "Matrix elements entered."))
```

A method for storing the information is needed. As the version of Golden Common LISP employed during the compilation of this book did not support multi-dimensional arrays, the data contained in the matrix is represented by three separate one-dimensional arrays. These store the elements of the left-hand column, middle column and right-hand column of the matrix separately and are called LC, MC and RC accordingly. SETF and MAKE-ARRAY are used three times at the beginning of INPUT-MATRIX to initialise these arrays, giving each three locations. This results in the total of nine array locations altogether, as required for the nine elements of the matrix.

The matrix will be related to the array locations in this regular pattern.

```
(AREF LC 0)     (AREF MC 0)     (AREF RC 0)

(AREF LC 1)     (AREF MC 1)     (AREF RC 1)

(AREF LC 2)     (AREF MC 2)     (AREF RC 2)
```

It is now particularly easy to fill the locations with the user's values via the single DOTIMES loop. First, TERPRI tidies the display with a fresh line. Then FORMAT T indicates which row of the matrix is to be entered. (PRIN1 (1+ X)) adjusts the value displayed, as the user will expect the first row to be 1 and not zero. Each of the three arrays can now be filled simultaneously, because the value of the loop variable conveniently matches the location in all three arrays. (SETF (AREF LC X) (READ)) assigns the user's input to the array storing the left-hand column of the matrix. The other two columns are filled identically.

Outside the DOTIMES loop, TERPRI and FORMAT T display a message which indicates that the matrix is now filled. Here is a typical application of INPUT-MATRIX, using the matrix given in the example above.

```
* (INPUT-MATRIX)
Row No.1
1 1 1
Row No.2
2 -3 1
Row No.3
2 -2 1
Matrix elements entered.
```

Calculating the Determinant - DETERMINANT

The first step in solving a set of equations is to calculate the determinant formed by the coefficients of the variables. This is quite a simple arithmetical procedure, but naturally care has to be taken when the coding into LISP is carried out. The main difficulty is in relating the usual format of the equations into the different order required in LISP.

In terms of the array elements chosen to store the matrix (initially using conventional arithmetical notation to aid understanding of what is taking place), the determinant can be expanded into the expressions:

```
   (AREF LC 0) x ( (AREF MC 1) x (AREF RC 2) - (AREF MC 2)
x (AREF RC 1) )

- (AREF MC 0) X ( (AREF LC 1) X (AREF RC 2) - (AREF LC 2)
x (AREF RC 1) )
+ (AREF RC 0) X ( (AREF LC 1) X (AREF MC 2) - (AREF LC 2)
x (AREF MC 1) )
```

Temporarily ignoring the sign of each expression, these are coded to give these s-expressions.

```
(* (AREF LC 0) (- (* (AREF MC 1) (AREF RC 2)) (* (AREF
 MC 2) (AREF RC 1))))

(* (AREF MC 0) (- (* (AREF LC 1) (AREF RC 2)) (* (AREF
 LC 2) (AREF RC 1))))

(* (AREF RC 0) (- (* (AREF LC 1) (AREF MC 2)) (* (AREF
 LC 2) (AREF MC 1))))
```

Observe that when writing expressions which closely resemble one another it is helpful if they are written down in a way which facilitates matching the equivalent brackets. This provides an easy check on whether the total numbers of opening and closing brackets are the same, assuming, of course, that the same mistake is not simply being repeated in each expression.

The symbolic-expressions can now be rearranged and combined, with appropriate addition and subtraction taking place, to give the function DETERMINANT. Again care must be taken in balancing the brackets when so much nesting of expressions is occurring.

```
(DEFUN DETERMINANT NIL
 (- (+ (* (AREF LC 0) (- (* (AREF MC 1) (AREF RC
          2)) (* (AREF MC 2) (AREF RC 1))))
       (* (AREF RC 0) (- (* (AREF LC 1) (AREF MC
          2)) (* (AREF LC 2) (AREF MC 1)))))
       (* (AREF MC 0) (- (* (AREF LC 1) (AREF RC
          2)) (* (AREF LC 2) (AREF RC 1))))))
```

The function can be tested upon the matrix already entered as an example in the previous section. The determinant for this matrix will be (see next page):

1 x ((-3 X 1) - (1 x -2)) - 1 x ((2 x 1) - (1 x 2))
+ 1 x ((2 x -2) - (-3 x 2))

This reduces to the value 1. Trying the new function gives:

```
* (DETERMINANT)
1
```

Using the function after entering other matrix elements with `INPUT-MATRIX` will give a further test that `DETERMINANT` does, indeed, calculate the determinant of a matrix.

Finding the Inverse - INVERSE-MATRIX

The inverse matrix is stored in a similar way to the original matrix. Three arrays are used, `INV-L`, `INV-M` and `INV-R`, representing the left, middle and right columns of the new matrix.

```
(AREF INV-L 0)   (AREF INV-M 0)   (AREF INV-R 0)

(AREF INV-L 1)   (AREF INV-M 1)   (AREF INV-R 1)

(AREF INV-L 2)   (AREF INV-M 2)   (AREF INV-R 2)
```

The function `INVERSE-MATRIX` has to calculate the values which are placed in these nine locations. The method for calculating the inverse of a matrix consists of four stages. For each element in the original matrix, the cofactor is first found. This is the determinant of the lesser rank matrix remaining when the row and column containing the element are deleted from the matrix. The second stage is to divide the cofactor by the determinant of the matrix. Then a new matrix is formed from the cofactors, but exchanging rows and columns. This means, for example, that the element in the second row and third column of the new matrix is the cofactor of the element in the third row and second column of the original matrix. Finally, a sign change is imposed upon half of the elements in the new matrix, forming an alternating, checkerboard pattern. Thus the top left-hand element does not undergo a sign change, but the element to its right does and also the one immediately below it. These rules are applied in the function `INVERSE-MATRIX`.

As an example, the value for the array location `(AREF INV-M 0)` will be calculated. This is derived from the cofactor of the location `(AREF LC 1)` in the initial matrix. This is because the position of `(AREF INV-M 0)` in its matrix is first row and second column, while the location of `(AREF LC 1)` is second row and first column. The transposing of rows and columns has thus taken place. The cofactor is therefore:

(- (* (AREF MC 0) (AREF RC 2)) (* (AREF MC 2) (AREF RC 0)))

This has to be divided by the determinant, stored in the variable DET, and also requires a change of sign. Making these two modifications, the value which needs to be assigned to the array location (AREF INV-M 0) becomes:

(/ (- (* (AREF MC 2) (AREF RC 0)) (* (AREF MC 0) (AREF RC 2))) DET)

This appears in the function, together with eight other similar expressions which are deduced in the same way. Before these are placed in the function, the three arrays have to be set up and the correct value assigned to DET. The function then has this form.

```
(DEFUN INVERSE-MATRIX NIL
 (SETF INV-L (MAKE-ARRAY 3))
 (SETF INV-M (MAKE-ARRAY 3))
 (SETF INV-R (MAKE-ARRAY 3))
 (SETF DET (DETERMINANT))
 (SETF (AREF INV-L 0) (/ (- (* (AREF MC 1) (AREF
  RC  2)) (* (AREF MC 2) (AREF RC 1))) DET))
 (SETF (AREF INV-M 0) (/ (- (* (AREF MC 2) (AREF
  RC  0)) (* (AREF MC 0) (AREF RC 2))) DET))
 (SETF (AREF INV-R 0) (/ (- (* (AREF MC 0) (AREF
  RC  1)) (* (AREF MC 1) (AREF RC 0))) DET))
 (SETF (AREF INV-L 1) (/ (- (* (AREF RC 2) (AREF
  RC  1)) (* (AREF LC 1) (AREF RC 2))) DET))
 (SETF (AREF INV-M 1) (/ (- (* (AREF LC 0) (AREF
  RC  2)) (* (AREF LC 2) (AREF RC 0))) DET))
 (SETF (AREF INV-R 1) (/ (- (* (AREF LC 1) (AREF
  RC  0)) (* (AREF LC 0) (AREF RC 1))) DET))
 (SETF (AREF INV-L 2) (/ (- (* (AREF MC 1) (AREF
  MC  2)) (* (AREF LC 2) (AREF MC 1))) DET))
 (SETF (AREF INV-M 2) (/ (- (* (AREF LC 2) (AREF
  MC  0)) (* (AREF LC 0) (AREF MC 2))) DET))
 (SETF (AREF INV-R 2) (/ (- (* (AREF LC 0) (AREF
  MC  1)) (* (AREF LC 1) (AREF MC 0))) DET)))
```

The function can be tested by displaying its value with a further function PRINT-INVERSE.

Displaying the Inverse Matrix - PRINT-INVERSE, GAP

In the actual solution of equations, the inverse matrix need never be displayed explicitly. It is simply a stage on the way to the final answer. It may be interesting, though, to see precisely how the solution is proceeding. Being able to check the inverse matrix is also a useful test to have available should unexpected results be produced by the overall function SOLVE-EQN. The function which displays the inverse matrix is PRINT-INVERSE.

In order to present a matrix neatly on the screen, PRINT-INVERSE makes use of another function, GAP. This will produce the spacing required to lay out columns of figures neatly, in a similar way to the earlier function LINE. Here is GAP coded.

```
(DEFUN GAP (N)
 (DOTIMES (X (- 7 (FLATC N)) T)
          (FORMAT T " ")))
```

The function takes an argument, N. This will represent any of the matrix elements. (FLATC N) will be the number of characters in N, including, for example, any minus sign. The loop will then use (FORMAT T " ") to print a number of spaces inversely proportional to the amount of room taken up by N. This can then be used to place numbers in regular columns on the screen. The choice of 7 as the maximum gap size is quite arbitrary and may be altered according to the numbers being produced as the equations are solved. As GAP is distinct from PRINT-INVERSE, this adjustment can be made very easily. Alternatively, GAP could have been written with an extra variable, W, on the parameter list:

```
(DEFUN GAP ( W N)
 (DOTIMES (X (- W (FLATC N)) T)
          (FORMAT T " ")))
```

A value for this would now need to be selected by the user and so the function PRINT-INVERSE would involve the extra lines:

```
(FORMAT T "Type spacing for matrix, 5-15:")
(TERPRI)
(SETF W (READ))
```

As it stands, PRINT-INVERSE does not have this option. This is how it is coded.

```
(DEFUN PRINT-INVERSE NIL
(INVERSE-MATRIX)
(DOTIMES (X 3 T)
         (TERPRI)
         (FORMAT T "1")
         (GAP (AREF INV-L X))
         (PRIN1 (AREF INV-L X))
         (GAP (AREF INV-M X))
         (PRIN1 (AREF INV-M X))
         (GAP (AREF INV-R X))
         (PRIN1 (AREF INV-R X))
         (FORMAT T "    1")))
```

First, the inverse matrix is calculated by simply calling the name of the appropriate function with (INVERSE-MATRIX). The DOTIMES loop then prints out the matrix

In order to present a matrix neatly on the screen, PRINT-INVERSE makes use of another function, GAP. This will produce the spacing required to lay out columns of figures neatly, in a similar way to the earlier function LINE. Here is GAP coded.

```
(DEFUN GAP (N)
 (DOTIMES (X (- 7 (FLATC N)) T)
          (FORMAT T " ")))
```

The function takes an argument, N. This will represent any of the matrix elements. (FLATC N) will be the number of characters in N, including, for example, any minus sign. The loop will then use (FORMAT T " ") to print a number of spaces inversely proportional to the amount of room taken up by N. This can then be used to place numbers in regular columns on the screen. The choice of 7 as the maximum gap size is quite arbitrary and may be altered according to the numbers being produced as the equations are solved. As GAP is distinct from PRINT-INVERSE, this adjustment can be made very easily. Alternatively, GAP could have been written with an extra variable, W, on the parameter list:

```
(DEFUN GAP ( W N)
 (DOTIMES (X (- W (FLATC N)) T)
          (FORMAT T " ")))
```

A value for this would now need to be selected by the user and so the function PRINT-INVERSE would involve the extra lines:

```
(FORMAT T "Type spacing for matrix, 5-15:")
(TERPRI)
(SETF W (READ))
```

As it stands, PRINT-INVERSE does not have this option. This is how it is coded.

```
(DEFUN PRINT-INVERSE NIL
(INVERSE-MATRIX)
(DOTIMES (X 3 T)
         (TERPRI)
         (FORMAT T "l")
         (GAP (AREF INV-L X))
         (PRIN1 (AREF INV-L X))
         (GAP (AREF INV-M X))
         (PRIN1 (AREF INV-M X))
         (GAP (AREF INV-R X))
         (PRIN1 (AREF INV-R X))
         (FORMAT T "    l")))
```

First, the inverse matrix is calculated by simply calling the name of the appropriate function with (INVERSE-MATRIX). The DOTIMES loop then prints out the matrix

produce the unit matrix. Hence a correct inverse has been calculated by INVERSE-MATRIX. It can now be applied to its specific role, that of obtaining the solution to a set of equations.

Linear Equations in LISP - SOLVE-EQN

The final function for solving equations is coded as SOLVE-EQN.

```
(DEFUN SOLVE-EQN NIL
 (SETF CV (MAKE-ARRAY 3))
 (SETF ANS (MAKE-ARRAY 3))
 (TERPRI)
 (FORMAT T "Type coefficients of variables:")
 (INPUT-MATRIX)
 (TERPRI)
 (FORMAT T "Type column vector:")
 (DOTIMES (X 3 T)
          (TERPRI)
          (FORMAT T "Row No.")
          (PRIN1 (1+ X))
          (TERPRI)
          (SETF (AREF CV X) (READ)))
(INVERSE-MATRIX)
(DOTIMES (X 3 T)
         (SETF (AREF ANS X) (+ (* (AREF INV-L X)
          (AREF CV 0)) (* (AREF INV-M X)
            (AREF CV 1)) (* (AREF INV-R X)
               (AREF CV 2)))))
 (TERPRI)
 (FORMAT T "Solution of equations is:")
 (TERPRI)
 (FORMAT T "X-value: ")
 (PRIN1 (AREF ANS 0))
 (TERPRI)
 (FORMAT T "Y-value: ")
 (PRIN1 (AREF ANS 1))
 (TERPRI)
 (FORMAT T "Z-value: ")
 (PRIN1 (AREF ANS 2))
 (TERPRI)
 (FORMAT T "Equations solved."))
```

Two extra arrays are introduced. CV holds the column vector representing the right-hand side of the three linear equations. ANS is used to record the three answers to the variables, the X, Y and Z values. These letters are assumed to be the ones chosen by the user for the equations. Then after the prompt 'Type coefficients of variables', INPUT-

MATRIX requests the numbers forming the matrix on the left-hand side. Another prompt, 'Type column vector' follows and then a DOTIMES loop which places the numbers given into the CV array. This is done by the expression (SETF (AREF CV X) (READ)). Additionally, TERPRI, FORMAT T and PRIN1 make the screen display easier to understand at this stage. The loop variable must be incremented by 1 before being printed on the screen. It would be confusing to refer to the first row with the numeral zero, which is the first value of X in the loop.

The values for the new matrix are calculated by (INVERSE-MATRIX). A second DOTIMES loop can now use the values in the three arrays, INV-L, INV-M and INV-R, to place values for the three variables into the ANS array. The two matrices which have to be multiplied to give the solution of the equations are these:

```
l(AREF INV-L 0) (AREF INV-M 0) (AREF INV-R 0)l
l                                             l
l(AREF INV-L 1) (AREF INV-M 1) (AREF INV-R 1)l
l                                             l
l(AREF INV-L 2) (AREF INV-M 2) (AREF INV-R 2)l

                        X

                l (AREF CV 0) l
                l             l
                l (AREF CV 1) l
                l             l
                l (AREF CV 2) l
```

Their product has to be placed in the answer array:

```
l(AREF ANS 0)l
l            l
l(AREF ANS 1)l
l            l
l(AREF ANS 2)l
```

Again the standard method for multiplying matrices is employed, working along a row of the inverse matrix and down the corresponding elements in the column matrix. It can then be seen that a typical element in the ANS array, (AREF ANS X), will be given by this expression:

```
(+ (* (AREF INV-L X) (AREF CV 0)) (* (AREF INV-M X) (AREF
CV 1)) (* (AREF INV-R X) (AREF CV 2)))
```

This appears in the DOTIMES loop. The two extra closing brackets at the end are for the SETF and the DOTIMES. The final part of the function just prints out the answers neatly. This section of SOLVE-EQN could be made more compact if the answers were

simply referred to by a number, rather than X, Y and Z. Another loop could be used to avoid repetition. The extra lines of the function do, nevertheless, make the screen display more explicit.

SOLVE-EQN can be tested with the set of equations stated initially.

```
* (SOLVE-EQN)
Type coefficients of variables:
Row No. 1
3 2 1
Row No.2
1 -1 3
Row No.3
2 1 1
Matrix elements entered.
Type column vector:
Row No. 1
5
Row No.2
6
Row No.3
4
Solution of equations is:
X-value: 2.0
Y-value: -1.0
Z-value: 1.0
Equations solved.
```

Summary

1. Quite ambitious mathematical LISP applications can be constructed using the minimum of 'core' LISP.

2. A hierarchical, top-down structure is essential for program clarity.

3. FORMAT T can be used to print text on the screen without brackets appearing.

4. TERPRI moves output to a fresh screen line.

5. FLATC returns the number of characters in an item.

6. NULL is a predicate function, which reverses the Boolean value of its argument.

7. Y-OR-N-P is useful in communicating with the user. It can be included in any specified prompt and returns T or NIL according to the user's response.

Exercise 4

State the value returned by each of the following.

1. `(NULL (< 9 4))`

2. `(NULL (= (* 3 4) (1+ 11)))`

3. `(NULL (EQUAL (REST '(A B C D)) '(B C D)))`

4. Use `FORMAT T` and `TERPRI` to construct a function which correctly prints the opening four lines of Richard III.

5. Use `Y-OR-N-P` in a `DO` loop to produce `SCROUNGE`, which keeps asking the user for money.

5

A La Recherché: Records, Trees and Networks

Introduction

Two ways have already been encountered of storing information within a LISP program. One method involved the arrays which Golden Common LISP supports and which were used extensively in the statistical functions illustrated in the previous two chapters. Another implicit method by which data has been held is the list itself. Both of these are examples of data structures available to the programmer, but they are not the only ways of recording information within LISP.

Two characteristic methods of storing data in LISP are the property list and the association list. Each provides a different way in which information needed for the operation of the program can be held until it is needed. Which of the two is most appropriately employed depends upon the particular application being investigated. An essential point to remember is that a data structure involves not simply the recording of information, but also the ease with which it may be accessed when required. This should be a guiding principle in choosing between the two.

It is also important to be conscious of the fact that LISP always allows the programmer considerable freedom to adapt the language to specific needs. New functions can be designed for particular applications, including the creation of new data structures. It will be shown how property lists and associations lists can be adapted to create new data structures like records, trees, networks and, in the following chapter, queues.

LISP Data Structure - the Property List

A property list is a direct way of recording information on a given subject. For example, a gardener might conceivably decide to make these notes on the forget-me-not, giving its colour, technical name and favoured soil type.

```
(COLOUR BLUE LATIN-NAME MYOSOTIS SOIL WELL-DRAINED)
```

If he did, this would be an example of a property list. Each property, 'blue', 'Myosotis' and 'well-drained', is linked with its respective indicator, 'colour', 'Latin-name' and 'soil'. More than one property may be specified by the indicators. This could be the property list for a wallflower.

(COLOUR (RED YELLOW WHITE) LATIN-NAME CHEIRANTHUS SOIL NON-ACID)

Here, the indicator COLOUR identifies a list of three colours, which are therefore included between brackets as a sublist.

Two functions exist in LISP to place information on to a property list and examine a list again to see its contents. These are PUTPROP and GET. A third, associated function is SYMBOL- PLIST.

Common LISP Primitive - PUTPROP

This function builds up a property list. Its general structure is like this.

```
(PUTPROP Symbol Property Indicator)
```

PUTPROP would be used in this fashion to attach the property NON-ACID to WALLFLOWER, with the indicator SOIL, as shown above.

```
* (PUTPROP 'WALLFLOWER 'NON-ACID 'SOIL)
WALLFLOWER
```

After this is typed, the property list for the symbol WALLFLOWER is (SOIL NON-ACID). Note how all the items have been quoted to prevent evaluation. The value returned by PUTPROP itself might at first be confusing.

Common LISP Primitive - GET

GET allows the property, or list of properties, identified by a particular indicator to be revealed. Its general structure is like this.

```
(GET  Symbol  Indicator)
```

It could be used after the above example to show the information that has been stored by PUTPROP.

```
* (GET 'WALLFLOWER 'SOIL)
NON-ACID
```

Common LISP Primitive - SYMBOL-PLIST

This function reveals the whole property list. If PUTPROP is used to assemble all the information given above for the symbol WALLFLOWER, then the property list can be retrieved in its entirety by employing (SYMBOL-PLIST 'WALLFLOWER). First, the property CHEIRANTHUS is added to the existing property NON-ACID.

```
* (PUTPROP 'WALLFLOWER 'CHEIRANTHUS 'LATIN-NAME)
WALLFLOWER
```

SYMBOL-PLIST will then show the current property list.

```
* (SYMBOL-PLIST 'WALLFLOWER)
(LATIN-NAME CHEIRANTHUS SOIL NON-ACID)
```

Adding the remaining items to the property list for WALLFLOWER is more complicated. A simple application of PUTPROP is not sufficient to add all of the colours, because only the most recent value will be retained. This can be seen happening in the following example.

```
* (PUTPROP 'WALLFLOWER 'WHITE 'COLOUR)
WALLFLOWER
* (SYMBOL-PLIST 'WALLFLOWER)
(COLOUR WHITE LATIN-NAME CHEIRANTHUS SOIL
 NON-ACID)
* (PUTPROP 'WALLFLOWER 'YELLOW 'COLOUR)
WALLFLOWER
* (SYMBOL-PLIST 'WALLFLOWER)
(COLOUR YELLOW LATIN-NAME CHEIRANTHUS SOIL NON-ACID)
```

The solution is not just to enter the extra colour with a direct use of PUTPROP, but instead to CONS the new colour on to a list of existing colours, returned by a direct use of GET. This requires in turn that the first colour added to the property list is NIL, because CONS always needs to start from NIL when building up a new list. Once NIL has initialised the list, CONS can build up the remaining elements as shown below. Of course, NIL does not reappear on the list after this.

```
* (PUTPROP 'WALLFLOWER NIL 'COLOUR)
WALLFLOWER
* (PUTPROP 'WALLFLOWER (CONS 'WHITE
    (GET 'WALLFLOWER 'COLOUR)) 'COLOUR)
WALLFLOWER
* (PUTPROP 'WALLFLOWER (CONS 'YELLOW
    (GET 'WALLFLOWER 'COLOUR)) 'COLOUR)
WALLFLOWER
* (PUTPROP 'WALLFLOWER (CONS 'RED
    (GET 'WALLFLOWER 'COLOUR)) 'COLOUR)
WALLFLOWER
```

SYMBOL-PLIST now reveals the property list originally intended.

```
* (SYMBOL-PLIST 'WALLFLOWER)
(COLOUR (RED YELLOW WHITE) LATIN-NAME CHEIRANTHUS
 SOIL NON-ACID)
```

The practical application of PUTPROP and GET in constructing an example of a record is now demonstrated.

Using a Property List - SIMPLEBASE

As an illustration of a simple database using records, a program will be written to show how a shopkeeper could monitor various items held in stock. The program developed will allow different categories of information to be stored about each individual item included in the database, although only four fields will be considered for the example. These are the item's name, the variety, the number held in the store and the sale price. A limited example of the stock held could be this.

Table 5.1

ITEM	VARIETY	NUMBER	PRICE
Beans	1.Standard 2.Curried	87	19
Icecream	1.Strawberry 2.Vanilla 3.Neopolitan 4.Raspberry-Ripple 5.Chocolate	59	25
Crisps	Salted	26	12

When this information is held in LISP, a typical property list will be (VARIETY (STANDARD CURRIED) NUMBER 87 PRICE 19).

The two obvious operations the shopkeeper will require for his rudimentary stocktaking software will be the choice of entering or accessing information held in the database. This is provided by the supervisory nature of the function SIMPLEBASE, which is able to call the working functions STORE and RETRIEVE. These will be developed afterwards. This again shows how top-down programming operates, with its ready assumption that the functions referred to can be written to fulfil their specified objectives.

```
(DEFUN SIMPLEBASE NIL
 (DO ((X 1 (1+ X)))
     ((> 0 1))
     (TERPRI)
     (FORMAT T "Store data - type S.")
     (TERPRI)
     (FORMAT T "Retrieve data - type R.")
     (TERPRI)
     (IF (EQUAL (READ) 'S) (STORE) (RETRIEVE))))
```

The entire function consists of a DO loop which does not exit, as a result of the impossible escape condition (> 0 1). As the main concern here is to demonstrate the practical use of PUTPROP and GET, this crude control structure is sufficient to allow repeated calls to the two functions STORE and RETRIEVE. At the end of this chapter a new conditional function will be introduced, which allows a more sophisticated approach to a similar program structure.

TERPRI and FORMAT T provide a clear screen display. The prompt suggesting that the user types the letter R is intended purely for psychological, or perhaps symmetrical, reasons. All the conditional expression detects is whether the letter S has been entered. If it has, followed either by the return key or simply by a space, the function STORE is evaluated. Any other key choice results in the function RETRIEVE.

Recording the Information - STORE

STORE uses PUTPROP to store the information typed into SIMPLEBASE.

```
(DEFUN STORE NIL
 (TERPRI)
 (FORMAT T "Type item, description, data.")
 (TERPRI)
 (SETF ITEM (READ))
 (SETF DESCRIPTION (READ))
 (SETF DATA (READ))
 (PUTPROP ITEM (CONS DATA (GET ITEM DESCRIPTION))
  DESCRIPTION))
```

TERPRI and FORMAT T neatly present a message requesting the item, description and data. These are then input by the use of READ and bound to the identifiers ITEM, DESCRIPTION and DATA. The way in which the information is placed on the property list is then directly analogous to the example given above. Of course, the identifiers do not have to be quoted.

The information can be obtained again with the function RETRIEVE. This function can be coded after another LISP primitive is introduced.

Common LISP Primitive - REVERSE

REVERSE takes a list as its argument and returns another list, the elements of which are in the opposite order.

```
* (REVERSE '(1 2 3 4 5))
(5 4 3 2 1)
* (REVERSE '(CHICKEN BEFORE EGG))
(EGG BEFORE CHICKEN)
```

Obtaining the Information - RETRIEVE

The function is coded as follows.

```
(DEFUN RETRIEVE NIL
 (TERPRI)
 (FORMAT T "Type item, description.")
 (TERPRI)
 (SETF L (REVERSE (GET (READ) (READ))))
 (DOTIMES (X (LENGTH L) T)
          (IF (> (LENGTH L) 1) (PRINT (1+ X))
           (TERPRI))
          (PRIN1 (NTH X L))))
```

The function begins with a request for the item being investigated and the specific field required. These two pieces of identifying information are placed immediately into the GET expression by READ and so assigned to L. However, because of the way the property list has been constructed by STORE, the last items of information will be the first on the list. This is not quite the way people are accustomed to seeing records presented and so, to make the display more agreeable, REVERSE is used to ensure 'last in, last out'.

A DOTIMES loop, related to the length of L, then presents the information. The conditional expression included within the loop, (IF (> (LENGTH L) 1) (PRINT (1+ X)) (TERPRI)), begins a new line on the display and numbers the items presented if there are more than one. What actually happens depends upon the value returned by the greater than predicate in (> (LENGTH L) 1). For more than one item, this will evaluate as T and so the first action takes place. The variable X is incremented by 1, because if it were not the first item would unconvincingly be numbered as zero. As PRINT is used here a new line of display is begun. For single items, the condition will return NIL. The action is then just TERPRI and so a new line is begun but no number added to the display. It is easy to include details like these in a function to increase greatly the clarity of the display produced.

Finally, each piece of information is placed on the correct line of the display, which will be the same as any number appearing, by the use of PRIN1 in (PRIN1 (NTH X L)).

A Practical Application of the Database

The database is illustrated here using some of the same information as given on the table.

```
* (SIMPLEBASE)
Store data - type S
Retrieve data - type R
S
Type item, description, data.
BEANS VARIETY STANDARD
Store data - type S
Retrieve data - type R
S
Type item, description, data.
ICECREAM VARIETY STRAWBERRY
Store data - type S
Retrieve data - type R
S
Type item, description, data.
CRISPS VARIETY PLAIN
Store data - type S
Retrieve data - type R
S
Type item, description.
BEANS NUMBER
87
Store data - type S
Retrieve data - type R
R
Type item, description.
CRISPS VARIETY
1 SALTED
2 PLAIN
Store data - type S
Retrieve data - type R
```

The response to a typical enquiry is:

```
R
Type item, description.
ICECREAM VARIETY
1 STRAWBERRY
2 VANILLA
3 NEOPOLITAN
4 RASPBERRY-RIPPLE
5 CHOCOLATE
```

Common LISP Primitive - REMPROP

The database here has no facility for full maintenance. Information can be added and retrieved but not deleted. Ideally, a further function should be incorporated into SIMPLEBASE as a third choice under the menu. Such a function would allow the removal of information which was no longer current. This additional function requires the new LISP primitive REMPROP.

REMPROP permits the removal of items from a property list. Its general structure is the same as that for GET.

```
(REMPROP symbol indicator)
```

In the following example, the property list for WALLFLOWER is progressively depleted by REMPROP. Each time, the existing list is shown first by SYMBOL-PLIST.

```
* (SYMBOL-PLIST 'WALLFLOWER)
(COLOUR (RED YELLOW WHITE) LATIN-NAME
 CHEIRANTHUS
SOIL NON-ACID)
* (REMPROP 'WALLFLOWER 'COLOUR)
WALLFLOWER
* (SYMBOL-PLIST 'WALLFLOWER)
(LATIN-NAME CHEIRANTHUS SOIL NON-ACID)
* (REMPROP 'WALLFLOWER 'LATIN-NAME)
WALLFLOWER
* (SYMBOL-PLIST 'WALLFLOWER)
(SOIL NON-ACID)
* (REMPROP 'WALLFLOWER 'SOIL)
WALLFLOWER
* (SYMBOL-PLIST 'WALLFLOWER)
NIL
```

The necessary function employing REMPROP will not be developed here. Instead, attention will be given to another data structure, the 'tree'.

LISP Data Structure - the Association List

Imagine there are three children, Cindy, Wendy and Mel, each of whom keeps pets. Cindy has a tortoise, Wendy a hamster and a gerbil and Mel a chihuahua and a macaw. A convenient way of recording the ownership of each animal would be to construct a symbolic-expression like this (see following page).

((MEL CHIHUAHUA MACAW) (WENDY HAMSTER GERBIL) (CINDY TORTOISE))

Each sublist shows the set of pets belonging to a particular child. Additionally, the child's name acts as an index to the sublist. This allows the possibility that the information may be more easily extracted from the overall list.

Such a structure does exist in LISP. It is called an association list, frequently abbreviated to a-list. Golden Common LISP provides several functions relating to association lists, including ACONS, which permits the creation of an association list, and ASSOC, which returns specified information from it.

Common LISP Primitive - ACONS

Several LISP dialects do not possess such a convenient function as Golden Common LISP's ACONS, designed specifically for the construction of an association list. ACONS has this overall form.

```
(ACONS  Key  Data  Association  list)
```

The key is the first element of each embedded sublist. It is used to identify the sublist when required. The data to be added to the association list follows and then the identifier for the association-list itself. Applied to the above example, and assuming that the identifier for the association list relating to the children and their animals is PETS, ACONS would be employed in this way.

```
(ACONS  'CINDY  '(TORTOISE)  PETS)
```

Note that the pet is being added as a list. Of course, if the association list is to be built up correctly, the new list has to be bound after each addition to the identifier PETS. In addition, PETS itself must be initially bound to NIL to avoid an unbound variable error from appearing. The complete list would be assembled in the following fashion.

```
* (SETF PETS NIL)
NIL
* (SETF PETS (ACONS 'CINDY '(TORTOISE) PETS))
     ((CINDY TORTOISE))
* (SETF PETS (ACONS 'WENDY '(HAMSTER GERBIL) PETS))
((WENDY HAMSTER GERBIL) (CINDY TORTOISE))
* (SETF PETS (ACONS 'MEL '(CHIHUAHUA MACAW) PETS))
* ((MEL CHIHUAHUA MACAW) (WENDY HAMSTER
     GERBIL) (CINDY TORTOISE))
```

Common LISP Primitive- ASSOC

This is the function which allows access to the data held in an association list. The arguments are the same as for ACONS, naturally without the data that was then being added, and the key to each sublist is again its first element. The general format of the function is therefore shown like this.

```
(ASSOC  Key  Association  list)
```

Applied to the association list PETS as constructed above, ASSOC allows each sublist to be returned in the following way.

```
* (ASSOC 'CINDY PETS)
(CINDY TORTOISE)
* (ASSOC 'WENDY PETS)
(WENDY HAMSTER GERBIL)
* (ASSOC 'MEL PETS)
(MEL CHIHUAHUA MACAW)
```

The key has to be quoted to prevent further evaluation, but the name of the association list must not be. Note how the entire sublist is returned, including the key itself at the head of the list.

In order to illustrate further their use, ACONS and ASSOC are now applied to a particular programming project.

Representing Information as a Tree

A tree is an important data structure. Its name is taken from the characteristic shape formed when its nodes and their linking arcs are drawn out in full. The structure then resembles an inverted tree. This similarity is due to the fact that although most of the nodes can have more than one child, no node ever has more than one parent. Hence the parents of all the nodes can be traced back up the tree to a single ancestor. This node is called the root. Moving down the tree, the eventual descendants of the root, and all the intermediate nodes, form a final set of nodes which have no children at all. The tree therefore ends plausibly with leaf nodes. It will be seen that the nomenclature employed for such a data structure leans heavily upon both genealogy and botany.

Tree structures emerge naturally in many applications in computer science, in the compilation of data structures, in administration, organisation of complex systems and particularly in artificial intelligence research for the attempted resolution of tasks represented in 'problem space'.

A typical tree structure could be shown as in Figure 5.1.

Binary Trees and Association Lists

In order to demonstrate the use of association lists in the representation of a tree structure, an example is now given of a simple binary tree in which it is assumed that there are only two possible arcs leading down from each node (see Figure 5.2). This tree will be used to hold the data needed for a simple guessing game. The objective of the game is to identify a town via a series of questions, each of which has a direct yes or no answer. Successive levels of nodes eliminate towns from the list of possible answers. Finally, the identity of the town is determined when the leaf nodes are reached.

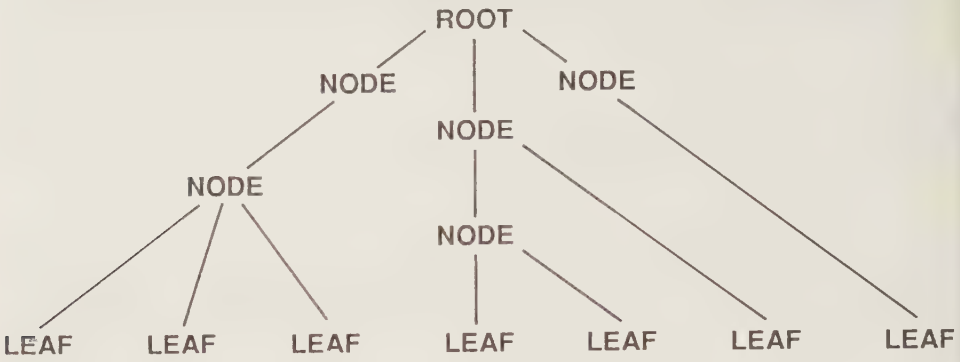


Figure 5.1

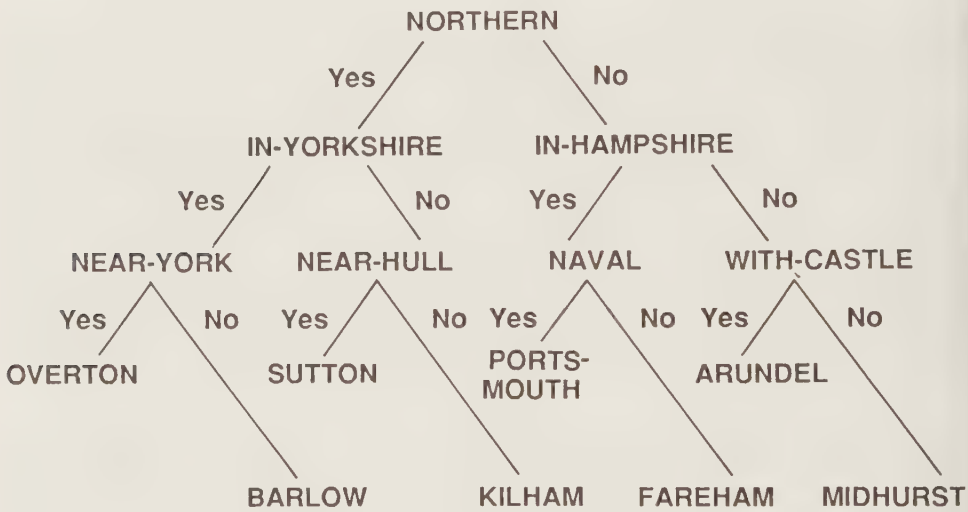


Figure 5.2

The tree structure will be represented as the association list `TREE`. Each sublist will have three elements. First, is the name of the node as the key to the sublist. Then follow the two children to which the node is linked. If these are referred to as `LEFT` and `RIGHT`, the general form of the list will be like this.

```
((NODE-1 LEFT-1 RIGHT-1) (NODE-2 LEFT-2 RIGHT-2) ...)
```

The advantage of using the association list to code the tree structure into LISP is that the children are readily obtained by using their parent node as the key to their location in the list.

First, a function has to be constructed which will allow the information to be placed into the association list. This function, called FILL-TREE, has been designed specifically for the list used here, TREE, but similar principles would apply in creating the simple routine for entering the nodes and arcs of the binary tree.

Entering the Nodes - FILL-TREE

The function is coded as follows.

```
(DEFUN FILL-TREE (N)
 (SETF TREE NIL)
 (TERPRI)
 (DOTIMES (X N T)
    (PRINT (1+ X))
    (FORMAT T "Type node, left branch, right
branch")
    (TERPRI)
    (SETF TREE (ACONS (READ)
    (LIST (READ) (READ)) TREE))) TREE)
```

The function has to be told how many nodes are to be filled for the tree. The way in which FILL-TREE is written means that the leaf nodes do not have to be included in this total. The DOTIMES loop then requests the content for each node, as well as the two nodes for which it is the parent. SETF and ACONS are employed in the expression which adds the new information into the a-list (SETF TREE (ACONS (READ) (LIST (READ) (READ)) TREE)). The arguments for ACONS take the standard form of key, data and association list, although the necessary use of READ may make this appear a little unclear at first. As evaluation begins at the lowest level of parenthesis and operates from left to right, the first instance of READ yields the key. The second and third use of READ are then combined by LIST into the list needed as the data. Finally, TREE, of course, is the identifier for the association list. Outside the loop, the evaluation of TREE before the last bracket allows the list's contents to be displayed on the screen as a check.
FILL-TREE would be used to enter the tree structure as follows.

```
* (FILL-TREE 7)
1 Type node, left branch, right branch
NORTHERN IN-YORKSHIRE IN-HAMPSHIRE
2 Type node, left branch, right branch
IN-YORKSHIRE NEAR-YORK NEAR-HULL
3 Type node, left branch, right branch
IN-HAMPSHIRE NAVAL WITH-CASTLE
4 Type node, left branch, right branch
NEAR-YORK OVERTON BARLOW
5 Type node, left branch, right branch
```

```
NEAR-HULL SUTTON KILHAM
6 Type node, left branch, right branch
NAVAL PORTSMOUTH FAREHAM
7 Type node, left branch, right branch
WITH-CASTLE ARUNDEL MIDHURST
((WITH-CASTLE ARUNDEL MIDHURST) (NAVAL PORTSMOUTH
 FAREHAM) (NEAR-HULL SUTTON KILHAM) (NEAR-YORK
  OVERTON BARLOW) (IN-HAMPSHIRE NAVAL WITH-CASTLE)
   (IN-YORKSHIRE NEAR-YORK NEAR-HULL) (NORTHERN
     IN-YORKSHIRE IN-HAMPSHIRE))
```

Retrieving the Data - LEFT, RIGHT, BRANCH

Functions now have to be written which will extract the necessary information from the association list when required. Fortunately, the way in which the list has been constructed makes it very easy to obtain the left and right children of any node, using the functions LEFT and RIGHT.

```
(DEFUN LEFT (NODE)
 (SECOND (ASSOC NODE TREE)))
```

In LEFT, (ASSOC NODE TREE) obtains the sublist (NODE LEFT-CHILD RIGHT-CHILD). Hence (SECOND (ASSOC NODE TREE)) gives LEFT-CHILD.

```
(DEFUN RIGHT (NODE)
 (SETF R T)
 (THIRD (ASSOC NODE TREE)))
```

In a similar way, the function RIGHT uses THIRD to return the RIGHT-CHILD for a given node. The reason for assigning the value T to R will be understood when the remaining three functions are examined.

The function BRANCH then employs LEFT and RIGHT in allowing the user to select the next appropriate node of the tree in response to a question.

```
(DEFUN BRANCH (NODE)
 (SETF R NIL)
 (TERPRI)
 (FORMAT T "Is it ")
 (PRIN1 (LEFT NODE))
 (PRIN1 '?)
 (TERPRI)
 (IF (Y-OR-N-P) (LEFT NODE) (RIGHT NODE)))
```

First, the value of R is set to NIL. (PRINT (LEFT NODE)) then shows the node which would result from an affirmative reply to the question displayed by FORMAT T.

The Y-OR-N-P predicate in the conditional expression which follows selects the left child if the user has typed Y, or the right child if N has been pressed. After this, R will still have the value NIL if the left node has been selected but T if the right was chosen. FORMAT T, PRIN1 and TERPRI are again used throughout the function to create a neat display.

Using the Binary Tree - SEARCH-TREE, REPLY

BRANCH can now be included in the overall function which searches the tree.

```
(DEFUN SEARCH-TREE (NODE)
 (DO ((X 1 (1+ X)))
 ((NULL (LEFT NODE)))
 (SETF NODE (BRANCH NODE))
 (IF R (REPLY) T)))
```

SEARCH-TREE begins with a specified node, as indicated by the parameter list. A DO loop then repeats until the current node reached has no left child, detected as (NULL (LEFT NODE)). This will mean that a leaf has been reached. At each pass through the loop a new value is assigned to NODE by the symbolic-expression (SETF NODE (BRANCH NODE)). In this way, the passage through the tree is controlled by the answers given by the user. The program always asks whether the left child should be chosen. When instead the user requests the right child, the value of R calls the function REPLY.

```
(DEFUN REPLY NIL
 (TERPRI)
 (FORMAT T "It must be ")
 (PRIN1 NODE))
```

This prints a message to show that the program is aware of the change in direction at the current node. SEARCH-TREE operates as shown below. Note how the root node has been specified for the beginning of the search. Of course, a search could begin lower down the tree.

```
* (SEARCH-TREE 'NORTHERN)
Is it IN-YORKSHIRE?
 (Y or N) Yes
Is it NEAR-YORK?
 (Y or N) No
It must be NEAR-HULL
Is it SUTTON?
 (Y or N) No
It must be KILHAM

* (SEARCH-TREE 'NORTHERN)
```

```
Is it IN-YORKSHIRE?
 (Y or N) No
It must be IN-HAMPSHIRE
Is it NAVAL?
 (Y or N) Yes
Is it PORTSMOUTH?
 (Y or N) Yes
```

Networks and Trees

A tree structure like the one which has been investigated has many applications. There are occasions, though, where the nature of the information being stored involves a much more complicated set of relations between individual items of data. To draw a further analogy with a tree, it is as if it has been decorated for Christmas. Streamers, tinsel and flex for coloured lights now link the various branches in an intricate way and the simple, branching structure has been lost. On the data structure model, as soon as a link is made between two nodes the tree has become a network.

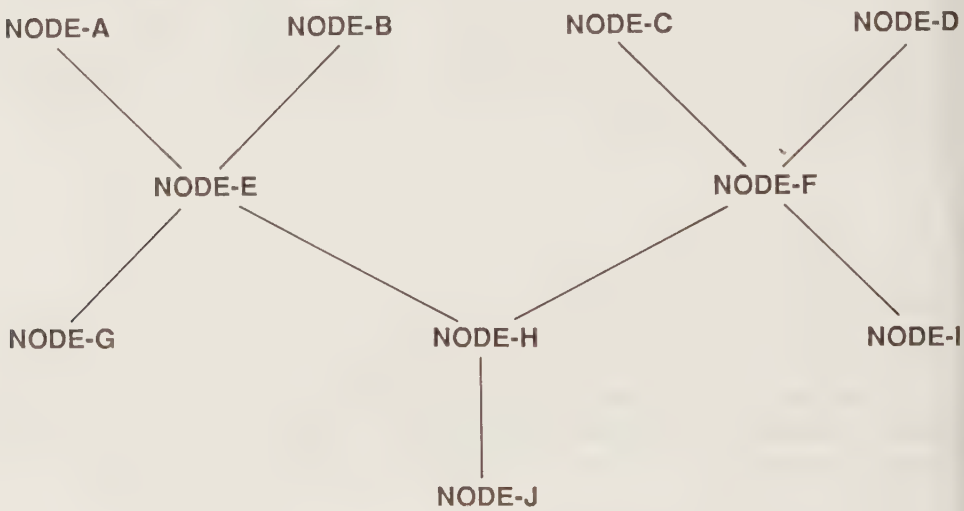


Figure 5.3

A network is therefore a more realistic data structure to consider employing when the data being handled requires additional, non-dendritic sideways links. Fortunately, the nodes and their linking arcs in a simple network can be easily represented as a diagram (see Figure 5.3). This, incidentally, makes clear the fundamental distinction between a network and a tree, because now it can be seen that in a network nodes do not necessarily have a unique parent. In Figure 5.3 NODE-H is the child of the two parent nodes, NODE-E and NODE-F.

The problem faced by the programmer is that of coding a network like this into LISP. As with any other data structure, the distinction has to be made between the mathematical nature of the relatively abstract structure itself and the particular way in which it is embodied by the programmer into a given language. Of course, the freedom presented by Golden Common LISP allows great latitude in the design of functions intended to store the information held by the network. Similarly, convenient routines can be developed which allow the user to obtain the information back again. Additionally, it is useful if the structure of the network itself is incorporated into the algorithms adopted, to permit automatic processing of some of the data held. Clearly, here, the transitive nature of the relationship between the nodes can be utilised. For example, if the nodes of the network represent prices of items in a shop and the arcs between them are to be interpreted as 'is more expensive than', then the individual pieces of information held in the network:

NODE-C is more expensive than NODE-F

NODE-F is more expensive than NODE-H

NODE-H is more expensive than NODE-J

inevitably lead to the conclusion:

NODE-C is more expensive than NODE-J

The way in which the information is held in the network leads to apparently intelligent reasoning. It is, admittedly, not quite equal to the work of the philosopher Aristotle. Nevertheless, the logic involved, the hypothetical syllogism, is sufficiently linked with his name to suggest the title of the next program.

Coding a Network into LISP - ARISTOTLE

`ARISTOTLE` is a program which will use a network to record various items of information and allow access to the data held in a variety of ways. It is, naturally, written as a series of individual LISP functions, each called when appropriate. In a typical, top-down programming style, it will be assumed that the functions being referred to do actually perform exactly as required and as specified in advance. Inevitably, top-down programming adopts this approach. The programmer has to declare that certain routines will operate, precisely as intended, when eventually they are written. Until then they exist simply as names, like 'black boxes' which accept the input they are given and produce the output required, without too much worrying about what goes on inside. Of course, this can lead to some embarrassed head scratching at two o'clock in the morning when a part of the code does not match up with expectations...

In `ARISTOTLE` the arcs between nodes represent the relation 'is'. This choice will not restrict the validity of any of the algorithms written and the program can, naturally, be adapted to other uses. It is, however, much simpler to visualise what is occurring in

various routines if concrete examples are provided of the content of each node and how they are related to one another. For this reason, the network is filled with simple items, all man-made artifacts sharing certain features (see Figure 5.4). It must be emphasised that the program will function with any information which can be matched to a network. The structure of the network and the data items placed at each of the nodes are both entirely arbitrary. They have simply been chosen to illustrate the various functions which will be developed.

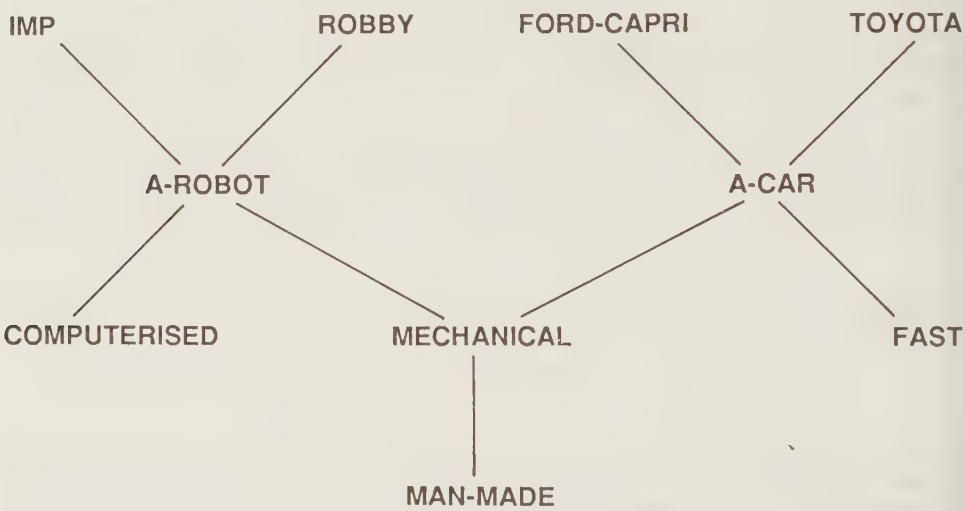


Figure 5.4

Two robots are deliberately featured in the network. Their presence is naturally justified since one of the most important goals of artificial intelligence research is surely the eventual development of machines resembling the robot of science fiction. And, as a corollary, LISP is the programming language which has been used to create most of the artificial intelligence software currently written.

A character note is needed. IMP, or Intelligent Mobile Platform, was a research robot built at Carnegie-Mellon University to investigate problems in automatic control of movement in a real, if indoor, environment. Similar in principle to the early robot, Shakey, at the Stanford Research Institute, it explored its surroundings using sonar. In contrast, Robby was entirely fictional. Unlike IMP, an unassuming metal cylinder approximating to the shape and height of a dustbin, Robby fitted the traditional image of the science fiction robot. He was man-sized and remarkably like a diving suit hybridised with a 1950s juke box, complete with clearly visible levers which tumbled noisily before each utterance. Robby first revealed himself to a group of astonished astronauts in the MGM film *'Forbidden Planet'* with the words, 'For your convenience I am monitored to respond to the name Robby'. Cyril Hume's script continued to give

Robby a consistently impressive command of natural language and a charming coincidence placed the film in the same year, 1956, as the Dartmouth Conference at which John McCarthy first proposed the term 'Artificial Intelligence'. It would be satisfying to believe that fiction had inspired potential fact...

The first function comprising ARISTOTLE is called ENTRY. As the title implies, it allows the input of information into the network, defining both the content of the nodes and the relations formed by the arcs linking them. First, however, two further LISP primitives will be explained, one which permits a more general form of branching within a program and the Boolean function AND.

Common LISP Primitive - COND

The IF function, which has been used exclusively until now, allows decisions to be made during the evaluation of a symbolic expression. It permits two courses of action, the 'then' and the 'else'. However, Common LISP, like all dialects of the language, also contains a function which allows more generalised branching. Here, a series of conditions are matched with their appropriate actions. The structure of a COND expression is like this.

```
(COND ((First condition) (First action))
      ((Second condition) (Second action))
      ((Nth condition) (Nth action))
      (T (Default action)))
```

Use of the COND function can involve the programmer in a more careful scrutiny of matching brackets. The outer set encompasses the overall COND. The middle sets of brackets define each separate if-then clause. Finally, the inner pairs delineate every individual condition or action.

When the COND is evaluated, each condition is considered in turn until one is found which evaluates to T. The action in the second part of the corresponding clause is then carried out and any further evaluation of the remaining clauses in the COND abandoned. Possibly none of the conditions will be true. In this case, the final clause of the COND has T as its condition and so the action associated with it will be what happens if all else fails.

A COND expression can be thought of as being rather like the bleary thoughts which might pass through an aging bachelor's mind when he first gathers consciousness in the morning and hits the mute button on the digital alarm.

```
(COND ((Popeye's on TVAM) (Switch the set on.))
((Wow. The Sun's appeared.) (Get the barbecue
 out.))
((It's Monday.) (I'll phone to say I'm sick.))
(T (That can't possibly be the time. Back to
 sleep!)))
```

A very simple COND is equivalent to the IF function already encountered. The two following pieces of code have the same effect.

```
(IF (Condition) (First action) (Second action))

(COND ((Condition) (First action))
(T (Second action)))
```

If operating speed is essential, then the IF form is more efficient.

Common LISP Primitive - AND

AND is to be regarded as a 'logical predicate'. Like all predicate functions it will always evaluate to T or NIL. It takes a series of arguments and returns NIL as soon as any one of them individually evaluates as NIL. It does not go on to consider the remaining arguments. However, if they all evaluate as T, AND returns T as well. It can be seen that this is equivalent to the common sense notion of 'and'.

```
* (AND (> 7 2) (= (* 4 2) 8))
T

* (AND (< 2 7) NIL)
NIL
```

The use of both COND and AND will be more clear after they have been illustrated in the function ENTRY.

Communicating with ARISTOTLE - ENTRY, REPEAT

The design of the 'front end' of even a piece of software as simple as ARISTOTLE is important. The person who is going to use a program is probably not its author and the little idiosyncrasies which a doting parent forgives are often the glaring character faults which society condemns. A minor quirk in the programmer's eyes could well be baffling incomprehensibilty for the user. LISP, in particular, will disenchant any ordinary member of the public if it demands all its discourse to invoke pairs of matching brackets. Essential as these are to the language, it is helpful if they can be kept internal to the program whenever possible, and not openly flaunted.

ENTRY is written to avoid such conspicuous parenthesis. Both the prompts presented by the computer, and the input accepted from the human user, are in conventional English. The former is achieved by using FORMAT T for the prompt, which, of course, prints on the display without using brackets. Also, by assembling the list E, using READ to acquire just the first three words typed by the user, brackets are avoided for the input as well.

```
(DEFUN ENTRY NIL
 (TERPRI)
 (FORMAT T "Statement, request, question,
 query?")
 (TERPRI)
 (SETF E (LIST (READ) (READ) (READ)))
 (COND ((EQUAL (SECOND E) 'IS) (STATEMENT))
       ((AND (EQUAL (FIRST E) 'DATA)
        (EQUAL (SECOND E) 'ON)) (REQUEST))
       ((EQUAL (FIRST E) 'IS) (QUESTION))
       ((AND (EQUAL (FIRST E) 'REASON)
        (EQUAL (SECOND E) 'WHY)) (QUERY))
       ((EQUAL (SECOND E) 'STOP) T)
       (T (REPEAT))))
```

TERPRI is used to place the prompt on a fresh line. FORMAT T then displays 'Statement, request, question, query?' This is intended to remind the user that four types of interaction with ARISTOTLE are possible.

The first is a statement of some information which must be added to the network. For example, FORD-CAPRI IS A-CAR could be a typical piece of information entered. It is in the form necessary if the program is to be able to accept the new data into the network. FORD-CAPRI and A-CAR both have to be defined as compound names with a hyphen. This makes them distinct entities and also ensures that the input is in the three word sentence structure required. Incidentally, more than three words are ignored by ENTRY. If the sentence FORD-CAPRI IS A-CAR AND IT IS REASONABLY FLASHY is typed in, the program will still assume that the user just intended FORD-CAPRI IS A-CAR.

The second way in which the user can address ARISTOTLE is by asking for everything it knows about a particular subject. This is done by typing sentences like DATA ON ROBBY or DATA ON TOYOTA. The first two words trigger the request function and the third indicates which node in the network must be investigated.

ARISTOTLE can also respond to a direct question of the form IS IMP COMPUTERISED. This is the third way in which the program will react to sentences typed in. If the user wishes to be grammatical and includes a question mark, a space must be placed between it and the final word. The function will then politely ignore it. A question mark directly following the third word without a gap will mean that it cannot be identified, because the punctuation has become part of the word itself. As will be explained below, ARISTOTLE replies with 'Relation not established' in circumstances like these.

Finally, a sentence of the form REASON WHY... will tell ARISTOTLE to provide a justification for the previous question it has answered. The program attempts to discover a logical link between various nodes of the network so that if it is asked IS

IMP MAN-MADE followed by REASON WHY YOU TOLD ME THAT (remember only the first three words matter), it will produce reasons for its answer. These consist of a chain of subjects and predicates following the general pattern of the example given above: IMP IS A-ROBOT, A-ROBOT IS MECHANICAL, MECHANICAL IS MAN-MADE.

ENTRY employs TERPRI a second time. Then the s-expression which follows, (SETF E (LIST (READ) (READ) (READ))), takes the first three words typed in on the keyboard and assembles them into a list bound to the identifier E. This is passed on to the COND expression. Here, the four types of entry are distinguished and the appropriately named functions, STATEMENT, REQUEST, QUESTION and QUERY, called as necessary. Four conditions are used to analyse the list E and select the function needed. The second and fourth of these are two-part conditions using AND.

STATEMENT is selected by the condition (EQUAL (SECOND E) 'IS), if the user has typed an entry of the form ROBBY IS A-ROBOT. If instead the sentence is like DATA ON A-CAR, the next condition, (AND (EQUAL (FIRST E) 'DATA) (EQUAL (SECOND E) 'ON)), will select the function REQUEST. The third condition, (EQUAL (FIRST E) 'IS), calls the function QUESTION whenever the input begins with IS. The fourth condition, which has been placed in the COND expression, (AND (EQUAL (FIRST E) 'REASON) (EQUAL (SECOND E) 'WHY)), will detect any entry beginning with the two words REASON WHY and call the function QUERY. Finally, the fifth condition in the COND, (EQUAL (SECOND E) 'STOP), accepts without further action any entry with STOP as the second word. The reason for this will be seen below.

The COND has an escape clause, the default action triggered by T if the words typed in by the user do not match any of the five acceptable forms. Sensibly enough, this action is the function REPEAT, which asks for another attempt at entry to be made.

```
(DEFUN REPEAT NIL
 (TERPRI)
 (FORMAT T "Please repeat")
 (ENTRY))
```

ENTRY can now be placed into ARISTOTLE to show how overall control of the program is organised.

ARISTOTLE in Action - the Network Assembled

As is typical in LISP, the main supervisory function is transparently simple. Here is the structure of ARISTOTLE.

(SETF NODES NIL) initialises the list which will hold all the individual node contents, such as COMPUTERISED, TOYOTA, FAST, etc. Then (SETF E NIL) is used to prevent an unbound variable error when the exit condition from the DO loop is first met. This loop repeatedly calls ENTRY to allow dialogue with the program. The escape condition it contains looks for STOP occurring as the second word in the input.

Thus a number of ways of leaving the program are possible. PLEASE STOP ARISTOTLE will work, or the more desperate STOP STOP STOP!

```
(DEFUN ARISTOTLE NIL
 (SETF NODES NIL)
 (SETF E NIL)
 (DO ((X 1 (1+ X)))
     ((EQUAL (SECOND E) 'STOP))
     (ENTRY))
     (TERPRI)
     (FORMAT T "You have left ARISTOTLE. Bye!"))
```

Until the function STATEMENT is written to allow information to be stored as a network, ARISTOTLE will not do anything. However, here is a preview of how the network information will be entered when STATEMENT is written. A temporary version of the function mimics the screen display which the proper coding of STATEMENT will produce.

```
(DEFUN STATEMENT NIL
 (TERPRI)
 (FORMAT T "Okay."))
```

The program will now allow this exchange:

```
* (ARISTOTLE)
Statement, request, question, query?
IMP IS A-ROBOT
Okay.
Statement, request, question, query?
ROBBY IS A-ROBOT
Okay.
Statement, request, question, query?
FORD-CAPRI IS A-CAR
Okay.
Statement, request, question, query?
TOYOTA IS A-CAR
Okay.
Statement, request, question, query?
A-ROBOT IS COMPUTERISED
Okay.
Statement, request, question, query?
A-ROBOT IS MECHANICAL
Okay.
Statement, request, question, query?
A-CAR IS MECHANICAL
Okay.
```

```
Statement, request, question, query?
A-CAR IS FAST
Okay.
Statement, request, question, query?
MECHANICAL IS MAN-MADE
Okay.
Statement, request, question, query?
PLEASE STOP NOW
You have left ARISTOTLE. Bye!
```

The complete version of STATEMENT cannot be developed until a new LISP primitive is introduced. This is the function MEMBER.

Common LISP Primitive - MEMBER

MEMBER takes as its arguments an atom and a list. If the atom is absent from the list, MEMBER evaluates as NIL. However, if the atom is present, MEMBER returns a sublist containing all the elements of the original list which follow the atom and including sublist includes the atom itself.

```
* (MEMBER 'F '(A B C D E))
NIL
* (MEMBER 'C '(A B C D E))
(C D E)
```

Recording the Data - STATEMENT, ADD-NODE, UPDATE, UNION, ARC

MEMBER can now be employed in the correct coding for STATEMENT. A number of other functions will also need to be defined, as indicated by the subtitle.

```
(DEFUN STATEMENT NIL
 (TERPRI)
 (FORMAT T "Okay.")
 (IF (MEMBER (FIRST E) NODES)
 (UPDATE) (ADD-NODE)))
```

The first part of the function is as before. It begins a new line of the display and places the message 'Okay' on the screen to show that the new information is being accepted. The conditional then checks if the subject of the sentence is already an element in the list NODES. If it is not, the function ADD-NODE is called to place the new node into the network.

```
(DEFUN ADD-NODE NIL
 (SETF NODES (CONS (FIRST E) NODES))
 (UPDATE))
```

Here, the subject of the sentence, (FIRST E), is added to the head of the list, NODES, with (CONS (FIRST E) NODES). This is then set up as the new list with (SETF NODES (CONS (FIRST E) NODES)). After this the function UPDATE is summoned.

UPDATE is encountered directly in the conditional expression in STATEMENT if (FIRST E) is a member of the list NODES. It examines the predicate of the sentence which has been typed and, if necessary, places it in the list of nodes. It also records the relation between the subject and predicate of the sentence in the network.

```
(DEFUN UPDATE NIL
 (SETF NODES (UNION (LIST (THIRD E)) NODES))
 (ARC (FIRST E) (THIRD E)))
```

The body of the function consists of immediate calls to two further functions, UNION and ARC. The first of these adds the predicate of the sentence entered by the user, (THIRD E), to the list NODES, without duplicating it if it is already present in the list. This could result if CONS were to be employed instead. The second function, ARC, codes the relationship between the subject and predicate as an arc in the network. Here is how the first of these functions is written.

```
(DEFUN UNION (LA LB)
 (DOTIMES (X (LENGTH LA) T)
          (IF (MEMBER (NTH X LA) LB) T
           (SETF LB (CONS (NTH X LA) LB)))) LB)
```

UNION takes two lists, LA, LB, as its arguments. This is why it was used in the form (UNION (LIST (THIRD E)) NODES) in UPDATE. It could be written in a simpler form which would accept an atom for its first argument. However, it could not then be used for a role it performs subsequently in the suite of functions composing ARISTOTLE. The DOTIMES loop uses (LENGTH LA) to control a search through the first list. MEMBER checks each element of LA in turn, (NTH X LA), for its presence in the second list, LB. If it is present, no further action is taken other than to return T. If the element is not already in LB, (MEMBER (NTH X LA) LB) will return NIL and the second action of the conditional expression is invoked. The list LB is then augmented by the element located in LA which it hitherto lacked. This is done in the s-expression (SETF LB (CONS (NTH X LA) LB)). When the DOTIMES loop has checked all the way through LA, the new list, LB, finally created consists of the union, in accordance with the familiar definition of set theory, of the two lists initially presented to the function.

The particular application of UNION in the function UPDATE is that if ARISTOTLE has been informed that A-CAR is mechanical and subsequently that MECHANICAL IS MAN-MADE, it will not attempt to redefine the list of nodes to include MECHANICAL twice. In fact, it would not affect the overall operation of the program if such duplication did occur. However, it would waste memory as well as being inelegant.

The second function used in UPDATE is ARC. This is a particularly crucial functio within ARISTOTLE because it is here, for the first time, that a decision has to be mad about the way in which the network will actually be represented in LISP. The obviou choice is between using property lists or association lists. The former is the on selected, allowing the immediate coding of the relationship between every subject an predicate. This is achieved by giving each node in the network a property calle CHILD. Then, if the sentence entered is A-ROBOT IS COMPUTERISED, an therefore the node A-ROBOT has to be linked to the node COMPUTERISED, the use o a property list with the expression (PUTPROP 'A-ROBOT 'COMPUTERISE 'CHILD) makes the necessary connection between the two. The link between the tw nodes is recalled, whenever required, by (GET 'A-ROBOT 'CHILD). Th generalised version of this is achieved by the function ARC.

```
(DEFUN ARC (NODE DATA)
 (PUTPROP NODE (CONS DATA (GET NODE 'CHILD))
  'CHILD))
```

The parameter list contains the variables NODE, representing the subject of the sentence entered, and DATA, representing the predicate. (GET NODE 'CHILD) finds the list of nodes with which NODE is already linked. (CONS DATA (GET NODE 'CHILD)) is the updated list of linked nodes and is immediately associated with NODE using (PUTPROP NODE (CONS DATA (GET NODE 'CHILD)) 'CHILD).

The functions ADD-NODE, UPDATE, UNION and ARC which STATEMENT calls therefore generate the network as information is typed in during the execution of ARISTOTLE. The other functions called by ENTRY, according to the user's choice, REQUEST, QUESTION and QUERY then allow the interrogation of the network. The first of these to be developed is REQUEST.

Viewing the Data - DISCLOSE, DESCENDANTS, IGNORANT, REQUEST

The second of the four main functions of ARISTOTLE, REQUEST, is now developed. It allows information to be obtained from the network when a request is made by the user. This could be, for example, DATA ON A-CAR. As the information is stored as a property list for each node, it is retrieved using GET. It is convenient to write a very simple function, DISCLOSE, first, which returns, as a list, all the nodes for which a particular node is the parent.

```
(DEFUN DISCLOSE (NODE)
 (GET NODE 'CHILD))
```

This function is then used in a further function, DESCENDANTS, which returns the children of a node, their children, their children's children... all the way to the leaf nodes of the network in a wholly Biblical fashion!

```
(DEFUN DESCENDANTS (NODE)
 (SETF DA NIL)
 (SETF DB (DISCLOSE NODE))
 (DO ((X 1 (1+ X)))
     ((EQUAL DA DB))
     (SETF DA DB)
     (SETF N (LENGTH DB))
     (DOTIMES (X N T)
              (SETF DB (UNION (DISCLOSE
               (NTH X DB)) DB)))) DB)
```

Essential to the operation of this function are the two lists, DA and DB. They are initialised outside the main DO loop in a way which ensures that they are not equal. This is vital if the escape condition of the loop, (EQUAL DA DB), is not to be immediately satisfied. However, once the condition is passed the two lists are equated by (SETF DA DB). The list DA can then determine the point at which the exit condition of the DO loop should be met and thus the stage when the nested DOTIMES loop, in which DB increases in length, should cease to be executed. This is when the list DB is no longer being augmented by further descendants of NODE.

The DOTIMES loop is controlled by the value of N, set to the length of DB outside the loop. Each element of DB is then considered in turn as (NTH X DB). DISCLOSE yields any children of the element and they are immediately added to the list by the expression (SETF DB (UNION (DISCLOSE (NTH X DB)) DB)). The use of UNION prevents any duplication in the new list. The DO loop keeps repeating the DOTIMES loop structure until DB has ceased to increase and so all the descendants of the original node are inevitably detected. DB is then placed before the final closing bracket, so that the function returns the complete list of descendants.

Before REQUEST can be finally assembled, one more function is needed. IGNORANT allows for the possibility that the information held in the network is insufficient to answer the user's question.

```
(DEFUN IGNORANT NIL
 (TERPRI)
 (FORMAT T "Sorry. No data available."))
```

Again, to make the screen display easier to read, TERPRI places the message on a fresh line. DESCENDANTS now allows the complete function REQUEST to be coded.

```
(DEFUN REQUEST NIL
 (DESCENDANTS (THIRD E))
 (IF (NULL DB) (IGNORANT) T)
 (DOTIMES (X (LENGTH DB) T)
          (PRINT (THIRD E))
          (PRIN1 'IS-)
          (PRIN1 (NTH X DB))))
```

As it is the third word of the sentence typed by the user which needs to be investigated, the symbolic expression employed by REQUEST is (DESCENDANTS (THIRD E)). When this is evaluated, the list DB created by DESCENDANTS contains all the nodes of the network which have (THIRD E) as their common ancestor. Of course, a question may have been asked about a node which has no descendants in the network at all. If this should happen, the conditional expression (IF (NULL DB) (IGNORANT) T) evaluates its first action and the function IGNORANT makes the apologetic reply 'Sorry. No data available.' If the node does have descendants, DB will not be NIL and so (NULL DB) returns NIL. The second action is then simply T. In either case, the DOTIMES loop is encountered because it lies outside the IF expression. If DB is empty, this is irrelevant. If there are elements in DB, though, all the descendants of the selected node are printed out neatly. This is done by judicious use of PRINT and PRIN1 in the loop to create a legible display. First, (PRINT (THIRD E)) begins a new line on the screen. PRINT creates a space after it, which means that (PRIN1 'IS-) follows after this first item in the correct position. The hyphen is needed because PRIN1 does not have a trailing space. (NTH X DB) is then added to the same line by a second use of PRIN1, rather than PRINT. Careful experimentation will always produce clarity of display if a little extra patience is contributed by the programmer.

ARISTOTLE can now be seen operating with the function REQUEST behaving as originally intended.

```
* (ARISTOTLE)
Statement, request, question, query?
DATA ON IMP
IMP IS-MAN-MADE
IMP IS-COMPUTERISED
IMP IS-MECHANICAL
IMP IS-A-ROBOT
Statement, request, question, query?
DATA ON FORD-CAPRI
FORD-CAPRI IS-MAN-MADE
FORD-CAPRI IS-MECHANICAL
FORD-CAPRI IS-FAST
FORD-CAPRI IS-A-CAR
Statement, request, question, query?
DATA ON SYLVIA
```

```
Sorry. No data available.
Statement, request, question, query?
ARISTOTLE STOP PREVARICATING
You have left ARISTOTLE. Bye!
```

Responding to a Specific Inquiry - QUESTION

The function DESCENDANTS makes it very obvious how to write the third main function QUESTION.

```
(DEFUN QUESTION NIL
 (TERPRI)
 (SETF SUB (SECOND E))
 (SETF PRED (THIRD E))
 (IF (MEMBER PRED (DESCENDANTS SUB))
  (FORMAT T "Yes.")
   (FORMAT T "Relation not established.")))
```

Once more, TERPRI improves the display. The two critical words of the sentence entered by the user are then recorded as the variables SUB and PRED, appropriate abbreviations for 'subject' and 'predicate'. For example, if the sentence typed is IS FORD-CAPRI FAST, then the value of SUB is FORD-CAPRI and the value of PRED is FAST. Doing this is not really essential for the operation of QUESTION itself, but the two words are needed again if the user next calls the function QUERY, as developed below.

If PRED is a direct or indirect property of SUB, then it will be one of its descendants and (MEMBER PRED (DESCENDANTS SUB)) will evaluate as a sublist, rather than NIL. As a non-NIL Boolean value is equivalent to T, this will cause the first action in the conditional expression to be evaluated. This is (FORMAT T "Yes.") and so the question has been given the affirmative reply required. But, if PRED is not a descendant of SUB, the expression (MEMBER PRED (DESCENDANTS SUB)) will be NIL and the second action of the conditional will be evaluated. The words 'Relation not established' appear on the screen. ARISTOTLE is now capable of exchanges of this nature:

```
* (ARISTOTLE)
Statement, request, question, query?
IS ROBBY COMPUTERISED
Yes.
Statement, request, question, query?
IS TOYOTA MAN-MADE
Yes.
Statement, request, question, query?
IS BOURBON FATAL
Relation not established.
Statement, request, question, query?
ARISTOTLE STOP NOW
You have left ARISTOTLE. Bye!
```

Before the last of the four main functions can be written, another piece of standard LISP code must be demonstrated.

Common LISP Predicate - DELETE

DELETE removes all occurrences of an element from a list.

```
* (DELETE 'A '(B A N A N A))
(B N N)
```

Automatic Reasoning - QUERY

The function DESCENDANTS is also used in the last of the four main functions, QUERY, which provides reasons for the answer given in a previous call to QUESTION. After QUESTION has been evaluated, the list DB contains all the descendants of SUB, one of which will be PRED, assuming that the answer to the question was affirmative. Some of these descendants will be on the direct route from SUB to PRED, but others will not be. To give a specific example, if the question had been IS IMP MAN-MADE then the list DB will contain all the elements A-ROBOT, COMPUTERISED, MECHANICAL, MAN-MADE. The elements A-ROBOT, MECHANICAL, MAN-MADE lie on the direct route from IMP to MAN-MADE, while COMPUTERISED does not. Quite clearly, nodes in the network not on the direct path between SUB and PRED must be ignored when providing a causal chain of reasoning which reflects the hypothetical syllogism in logic.

Another problem to be tackled is that the list of descendants will be complete all the way to the leaf nodes of the network, perhaps passing PRED on the way. Thus, if the question typed were IS TOYOTA MECHANICAL, the presence of MAN-MADE in the list of descendants of TOYOTA would be quite irrelevant. The elimination of such redundant nodes in the list of descendants is important for the algorithms used in QUERY. Such a reduction of the relevant nodes is so obvious to the human mind that it is done almost unconsciously. But, typically, a program needs to be told exactly how to cope with all possible eventualities in accordance with the familiar 'idiot savant' behaviour of computers.

There are therefore two types of node which need to be removed from the list of descendants, those not on the direct route to PRED and those which are lower than PRED in the network. QUERY manages to delete both kinds and then prints out the remaining nodes in their correct relation to one another.

```
(DEFUN QUERY NIL
 (SETF PATH (MEMBER PRED DB))
 (DOTIMES (X (LENGTH PATH) T)
          (IF (MEMBER PRED (DESCENDANTS
           (NTH X PATH))) T
             (DELETE (NTH X PATH) PATH)))
 (SETF ROUTE (REVERSE PATH))
```

```
(PRINT SUB)
(DOTIMES (X (1- (LENGTH PATH)) T)
         (PRIN1 'IS-)
         (PRIN1 (NTH X ROUTE))
         (PRINT (NTH X ROUTE)))
(PRIN1 'IS-)
(PRIN1 PRED))
```

First, QUERY uses (MEMBER PRED DB) to eliminate all the nodes beneath PRED in the network from DB. Fortunately the way in which DESCENDANTS was written ensures that the lowest nodes come first in the list of elements comprising DB. This new list of nodes is bound to the identifier PATH.

The first DOTIMES loop then examines all the nodes in PATH to remove any which are not on the direct route from SUB to PRED. Each node in the list will be identified in turn as (NTH X PATH). If such a node is on the direct route to PRED, then PRED will be a descendant of the node. This is coded into LISP as the condition (MEMBER PRED (DESCENDANTS (NTH X PATH))). If this symbolic-expression returns T, PRED is a descendant of the node, which can therefore be left in PATH. This results from the empty first action of T. However, if the condition evaluates as NIL then PRED is not a descendant of the node and it must be removed from PATH with (DELETE (NTH X PATH) PATH).

At the end of the loop only relevant nodes remain in PATH. They are in exactly the opposite order to the one required and so a new list is created, with (SETF ROUTE (REVERSE PATH)). All that is now needed is to print out the results. First, the subject of the original question is displayed with (PRINT SUB). After this there is a DOTIMES loop, one less in length than the number of nodes in ROUTE because of the inevitable duplication of the nodes in the mandatory arrangement of the display. PRIN1 is then used twice at each pass through the loop to print IS- and the appropriate node, (NTH X ROUTE). The same node is then displayed again, this time as the first word in the next line of the display, with PRINT creating this fresh line. Finally, the end of the route is shown with (PRIN1 PRED).
QUERY is used in ARISTOTLE after a call to QUESTION. It makes the program appear reasonably intelligent. ARISTOTLE will nevertheless work with the assumption that any use of QUERY refers to the last entry dealt with by QUESTION. In consequence, it can give some very silly responses, as shown in the second example below.

```
* (ARISTOTLE)
Statement, request, question, query?
IS FORD-CAPRI MAN-MADE
Yes
Statement, request, question, query?
REASON WHY THAT ANSWER GIVEN
FORD-CAPRI IS-A-CAR
```

```
A-CAR IS-MECHANICAL
MECHANICAL IS-MAN-MADE
Statement, request, question, query?
FAST IS DANGEROUS
Okay.
Statement, request, question, query?
REASON WHY FAST IS DANGEROUS
FORD-CAPRI IS-A-CAR
A-CAR IS-MECHANICAL
MECHANICAL IS-MAN-MADE
Statement, request, question, query?
PLEASE STOP AGAIN
You have left ARISTOTLE. Bye!
```

The example of a network described in this section has deliberately concentrated upon a very simple set of relationships, although the functions developed can be applied to a much wider set of related data.

Summary

1. An important way in which information can be stored in LISP is to use property lists.

2. Information is placed on a property list by PUTPROP and retrieved by GET.

3. SYMBOL-PLIST shows the whole property list.

4. REMPROP removes information from a property list.

5. Association lists are constructed by ACONS and information retrieved by ASSOC.

6. REVERSE inverts the order of a list's elements.

7. A more general form of branching is provided by COND.

8. AND is a logical predicate.

9. MEMBER determines whether an element belongs to a list.

10. DELETE removes an element completely from a list.

11. LISP allows the user great freedom in designing a data structure appropriate to a particular problem. Records, trees and networks are possible models to use.

Exercise 5

State whether T or NIL are returned by each of the following.

1. `(AND (= 5 5) (> 7 2) (< 3 9))`

2. `(AND (NULL (= (* 2 7) (/ 26 2))) (> 5 4) T)`

3.
```
(AND (NULL (MEMBER 'A '(B C D))) (EQUAL '(C D)
     (MEMBER 'C '(B C D))) (= 5 (1 - 6)))
```

4. Use COND to construct a simple function, LEADER, which states the Parliamentary party led by the politician whose name is entered by the user. For example, typing NEIL should give the response `'HE LEADS THE LABOUR PARTY'`.

5. Use COND to write a function which multiplies a number by 5 if it is greater than 7, subtracts 2 if it equals 3, but which adds 23 to it otherwise.

6. Write a short function which uses a property list to make a record of favourite TV programmes for the days of the week.

6

Minding Your Trees and Queues: Further Data Structures

Introduction

As its name suggests, a queue is a data structure which resembles the familiar line of people seen waiting for a bus. In fact, an appropriate mathematical model for the prospective passengers is, formally, a queue. In this the significant characteristics are that all new 'information' being stored, each person joining the queue, is added as items at one end of a linear list and that processing of the information occurs as individual reception of the 'items', passengers paying the fare, at the other. The acronym for the process is First In, First Out. This leads to 'FIFO list' as an alternative name which can be used for a queue.

Queueing theory has found wide application in many areas where the fundamental behaviour of a system under consideration involves arriving, waiting and eventual servicing. Apocryphally, it has even been remarked that British computer users, reflecting their supposed national character, should be the last to experience any difficulty in understanding related theory. In the vacuous example already cited, these three parts of the queueing model are simply the familiar experiences of running and simultaneously looking at your watch, waiting in the rain while staring at the traffic in the hope that a friend drives by, then finally arguing with the driver over whether the bus company has previously demanded exact fares. Naturally, more significant applications do exist. The mathematical properties of queues have been investigated since the initial work of Erlang in the second decade of this century and have found an essential role in the solution of many industrial problems. Ironically, despite the profusion of applications of queueing theory, total consensus still does not seem to exist over the spelling of its name. 'Queueing' and 'queuing' are both likely to appear in the literature.

The first use of a queue here will be in designing a systematic search of the type of tree structure already encountered in the previous chapter. The complexity which a tree may acquire as various nodes are added can make searching for a specific item very involved. Fortunately, there is a recognised method of assigning the nodes met during the search to a queue data structure. The way in which the queue is constructed allows

two separate approaches to a comprehensive examination of all the nodes of the tree. Before this method can be illustrated, however, a system is required for building up a more generalised tree structure than the one described earlier.

A General Tree Structure

In the last chapter a 'binary' tree was investigated. In a binary tree each node, except the leaves, has exactly two children. Binary trees are an important data structure, but it is also necessary to consider the type of structure in which the number of nodes linked to a parent is not limited to just two. In order to develop algorithms for building, and then searching, such a tree, the particular example shown in Figure 6.1 will be employed. The nodes of the tree are represented simply by letters of the alphabet, but in a real application it is assumed that significant information would be stored in these locations of the tree.

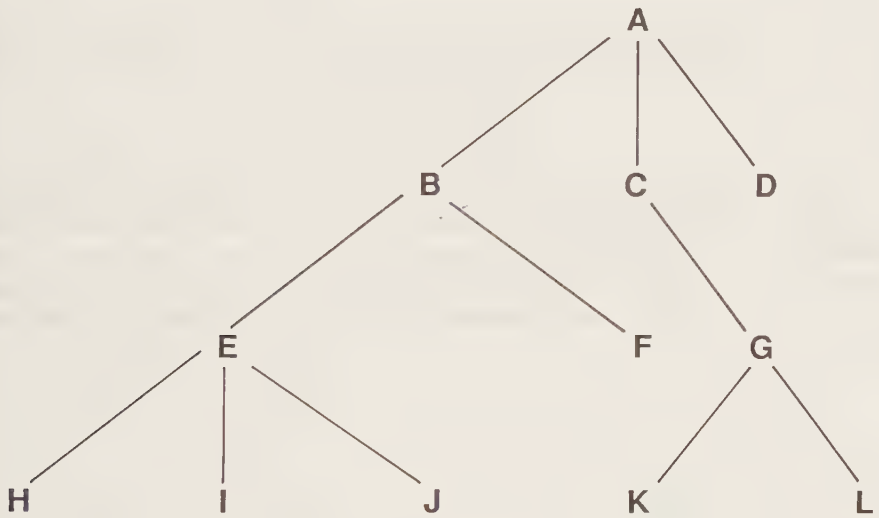


Figure 6.1

The root node is A. The leaf nodes are H, I, J, K and L. The root node has children B, C D. Node E also has three children, H, I and J. Nodes B and G have two children each, E, F and K, L. Node C only has one child, G. Nodes D and F have no children at all. Although this tree consists of only 12 nodes altogether, it possesses a structure which is sufficiently complicated to allow the development and testing of several useful algorithms, both in the construction of the tree itself and also in its subsequent searching. These would then be equally applicable to larger and more involved structures.

The first function which has to be written is, of course, the one which allows the tree to be coded into LISP. Before this function, BUILD-TREE, is can be created, however, there is another LISP primitive which must be added to those already in regular use.

Common LISP Primitive - NOT

NOT operates like NULL in reversing the Boolean value of a symbolic-expression. If the previous value of the expression were T, NIL will be returned for the new expression involving NULL. Similarly, NOT will change an expression evaluating as NIL into one which gives T. It proves to be extremely useful in the design of many algorithms. NOT is illustrated in the following two examples.

```
* (NOT T)
NIL

* (NOT (= 9 2))
T
```

In the first example, the value of T is simply replaced by NIL. In the second, the expression (= 9 2) evaluates as NIL and so the new expression (NOT (= 9 2)) has the value T.

Using Property Lists to Represent a Tree - BUILD-TREE

The previous function for constructing a tree, FILL-TREE, used association lists to record the relationship between nodes. Here, it is shown instead how property lists can represent the arcs between the nodes of the tree. For each node, any children are placed on to its property list with appropriate use of PUTPROP in the further function ADD-CHILD.

```
(DEFUN BUILD-TREE NIL
 (SETF NODE NIL)
 (DO ((X 1 (1+ X)))
     ((EQUAL NODE 'STOP))
     (TERPRI)
     (FORMAT T "Type node or STOP")
     (TERPRI)
     (SETF NODE (READ))
     (IF (NOT (EQUAL NODE 'STOP)) (ADD-CHILD) T))
 (TERPRI)
 (FORMAT T "Tree complete."))
```

First, (SETF NODE NIL) is used to prevent an unbound variable error occurring when the exit sequence is met for the first time. The DO loop which follows executes continually until the user explicitly instructs the program to end by typing STOP. The loop will then exit and the final TERPRI and FORMAT T place the explanatory message 'Tree complete' on the next line of the display.

Until that happens, the user is able to type in all the data needed to create any tree structure required, unlike FILL-TREE, which assumed a binary tree was being constructed. Within the loop, TERPRI and FORMAT T place the message 'Type node or STOP' on the screen. The user's choice is bound to the identifier NODE. If STOP has not been typed, (EQUAL NODE 'STOP) will evaluate as NIL and thus the overall condition (NOT (EQUAL NODE 'STOP)) will have the value T. This will result in the conditional expression selecting the function ADD-CHILD. Thus ADD-CHILD is called when the user has typed in the name of a node being joined on to the tree structure. The extra function is coded as follows.

```
(DEFUN ADD-CHILD NIL
 (SETF DATA NIL)
 (DO ((X 1 (1+ X)))
     ((EQUAL DATA 'NEXT))
     (TERPRI)
     (FORMAT T "Type child or NEXT")
     (TERPRI)
     (SETF DATA (READ))
     (IF (NOT (EQUAL DATA 'NEXT)) (PUTPROP NODE
     (CONS DATA (GET NODE 'CHILD)) 'CHILD) T)))
```

The structure of ADD-CHILD is analogous to that of BUILD-TREE. The new variable, DATA, is initially bound to NIL by (SETF DATA NIL) to prevent premature exit from the DO loop caused by any residual value from an earlier use of the function. The user is presented with the message 'Type child or NEXT' and the response bound to DATA. Every child of the current node can be entered by typing its name. When there are no further children to be added for the node, typing NEXT will trigger the exit condition, (EQUAL DATA 'NEXT). Return then occurs to the main function, BUILD-TREE, where the name of the next another node can be chosen or the program ended altogether.

If the name of the child is typed, however, the condition at the end of the loop, (NOT (EQUAL DATA 'NEXT)), will have the value T and so the first action occurs. This is precisely the same as the example given in the previous chapter. Any existing children of the current node are returned on its property list by the expression (GET NODE 'CHILD). Of course, if the node has only just been included in the tree, this property list will be NIL. The new list including the extra child is constructed by the use of CONS in (CONS DATA (GET NODE 'CHILD)) and installed as the current property list by the use of PUTPROP with the indicator 'CHILD.

The way in which the two functions can be used to enter the information needed to represent the original example given of a tree is shown below. The screen display indicates clearly to the user whether new nodes are being entered or the children of a previous node.

```
* (BUILD-TREE)
Type node or STOP
A
Type child or NEXT
D
Type child or NEXT
C
Type child or NEXT
B
Type child or NEXT
NEXT
Type node or STOP
B
Type child or NEXT
F
Type child or NEXT
E
Type child or NEXT
NEXT
Type node or STOP
E
Type child or NEXT
J
Type child or NEXT
I
Type child or NEXT
H
Type child or NEXT
NEXT
Type node or STOP
C
Type child or NEXT
G
Type child or NEXT
NEXT
Type node or STOP
G
Type child or NEXT
L
Type child or NEXT
K
Type child or NEXT
NEXT
Type node or STOP
STOP
Tree complete.
```

It must be emphasised, of course, that other trees can be entered with BUILD-TREE. Although only letters have been used to represent each node here, whole words could be placed at each node equally well.

Tree Structure and Search Techniques

Now that a function has been developed which will permit the construction of any arbitrary tree structure, attention can be given to precise methods of searching a tree for a required node. As mentioned above, the algorithms used in this organised form of search will involve the use of queues.

There are various ways in which a tree could be examined to extract desired information. The least interesting would probably be a random process in which nodes were selected without any overall guiding principle until the correct one was accidentally found. Such a 'dumb' search is scarcely worth considering further, though it is often observed in real life. Many people appear to select fiction books in a library, or clothes in a sale, in this fashion. A slightly better approach would be to consider all the nodes in turn, without repeating any of them. This 'blind' search would eventually locate the desired node, but it could scarcely be regarded as an intelligent approach to the problem.

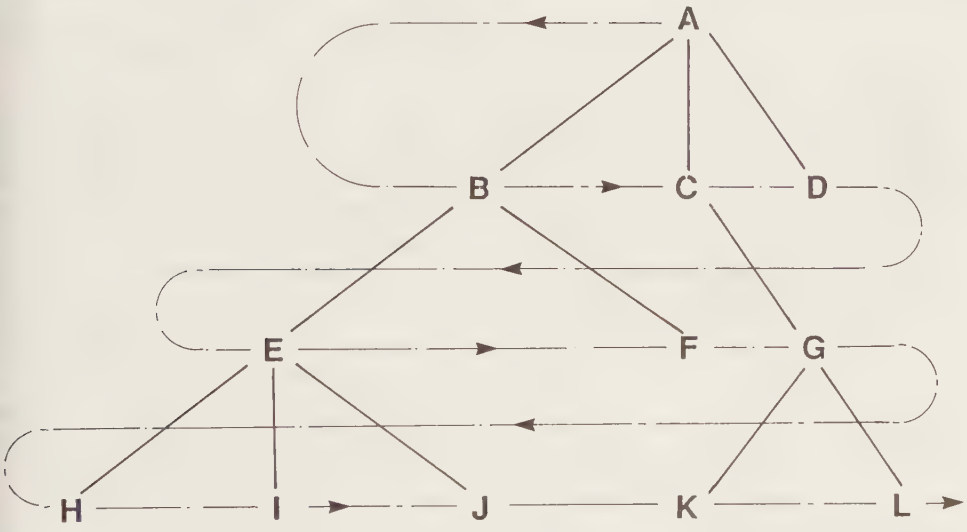


Figure 6.2 'Breadth-first' search

There are two standard methods of organising a search which will be developed in this chapter. Both permit a systematic investigation of the contents of a tree structure and

contribute significantly in many practical applications of problem solving. They each consider the nodes of a tree in a distinctive order. One method is termed a 'breadth-first' search. In this, all of the nodes at a particular level in a tree are examined before the next level is entered. Using the example of the tree already given, the order in which a breadth-first search of the nodes would take place is traced in Figure 6.2 by the arrow.

The order in which nodes are reached is A, B, C, D, E, F, G, H, I, J, K and L. The fact that this corresponds with alphabetical order is, of course, simply an accidental result of the way the nodes have been lettered and possesses no greater significance.

The order in which nodes are examined is quite different when a 'depth-first' search is employed. Here, the complete route to each leaf node is explored throughly before the searching process 'backs up' the tree to tackle the next route down to a leaf. Again, in Figure 6.3, the arrow shows the order in which nodes are checked.

The nodes are now located for the first time in the sequence A, B, E, H, I, J, F, C, G, K, L, D.

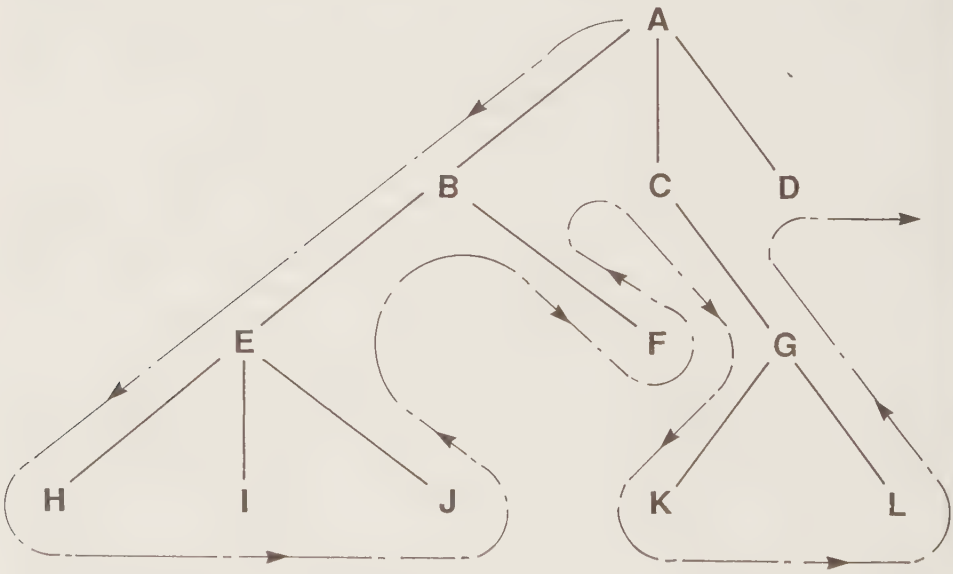


Figure 6.3 'Depth-first' search

A Useful Search Algorithm

There is a standard algorithm available which allows the design of either a breadth-first or depth-first search in an extremely elegant and simple fashion. In this a queue is

employed to decide the order in which nodes are considered. The first item in the queue is always the next node to be visited, but the vital part of the algorithm is the way in which the queue is updated each time a node is found to possess children which need to be included in the search.

If a breadth-first search is required, the new nodes are added at the end of the queue. In the example, this means that when the search begins at node A the queue is (B C D). Node B is then visited and as it has E and F as children these are added to the queue to give (C D E F). Naturally, node B itself has been removed from the head of the queue. Node C is the next to be investigated and its child, node G, is placed at the end of the queue. This then becomes (D E F G). It can be seen that the nodes are going to be located in the correct, breath-first order.

Alternatively, when a depth-first search is required, the additional nodes are placed at the head of the queue. Using the same tree for the search, beginning again at node A, will lead to an initial queue (B C D), as before. However, when node B is visited, its children are placed before the other nodes to yield the new ordering (E F C D). Similarly, after E has been checked the queue becomes (H I J F C D). Following this process for each node as it is reached will inevitably lead to a depth-first search.

Naturally, as a queue giving the order of the nodes is simply a list, it can be coded into LISP without any difficulty, as will be seen in the two functions to be developed, `BREADTH-FIRST` and `DEPTH-FIRST`.

Searching a Tree - BREADTH-FIRST

This function uses the algorithm described above to create a breadth-first search of any tree structure which might be established by the earlier function `BUILD-TREE`. In a practical application of the algorithm, it would be necessary to include code which recognised a particular piece of information required from the tree. This function, in contrast, is intended to be purely illustrative and so it simply states the identity of each node as it is reached. In addition, the current queue is also printed out on the display to show that the algorithm is behaving as expected. It would, of course, be quite easy to adapt the function to be part of a versatile data-handling routine.

```
(DEFUN BREADTH-FIRST NIL
 (TERPRI)
 (FORMAT T "Type starting node.")
 (TERPRI)
 (SETF L (LIST (READ)))
 (DO ((X 1 (1 + X)))
     ((NULL L))
     (TERPRI)
     (FORMAT T "Node visited No.")
     (PRIN1 X)
     (FORMAT T " is "
```

```
(PRIN1 (FIRST L))
(TERPRI)
(SETF L (REST (APPEND L (GET (FIRST L)
 'CHILD))))
(FORMAT T "Current queue is ")
(PRIN1 L)))
```

First, TERPRI and FORMAT T are used to request the node where the search should begin. SETF is used in the symbolic expression (SETF L (LIST (READ))) to bind this node as a list to the identifier L. L subsequently develops into the queue which determines the order in which the nodes are searched. The search itself takes place within the DO loop which follows. The exit condition from the loop is, quite simply, (NULL L). This is because when the queue of nodes to be checked has been exhausted the search must necessarily be over.

The variable which controls the loop is also used, together with FORMAT T and PRIN1, to number each node as it is reached. This makes the operation of the function easier to understand when the display appears on the monitor. The current node is, of course, the first item in L and so this is printed out by the expression (PRIN1 (FIRST L)).

After this, the queue must be updated by the appropriate method for a breadth-first search. This is done by obtaining the children of the current node with (GET (FIRST L) 'CHILD). These additional nodes are placed at the end of the queue by the expression (APPEND L (GET (FIRST L) 'CHILD)). Then the first node must be removed from the queue by using REST and the modified list bound again to L. This leads to the overall expression, (SETF L (REST (APPEND L (GET (FIRST L) 'CHILD)))). A final FORMAT T displays the current queue, so that the algorithm can be seen to be working.

The function is here shown is shown operating upon the same example of a tree.

```
* (BREADTH-FIRST)
Type starting node.
A
Node visited No.1 ia A
Current queue is (B C D)
Node visited No.2 is B
Current queue is (C D E F)
Node visited No.3 is C
Current queue is (D E F G)
Node visited No.4 is D
Current queue is (E F G)
Node visited No.5 is E
Current queue is (F G H I J)
Node visited No.6 is F
```

```
Current queue is (G H I J)
Node visited No.7 is G
Current queue is (H I J K L)
Node visited No.8 is H
Current queue is (I J K L)
Node visited No.9 is I
Current queue is (J K L)
Node visited No.10 is J
Current queue is (K L)
Node visited No.11 is K
Current queue is (L)
Node visited No.12 is L
Current queue is NIL
NIL
```

Searching a Tree - DEPTH-FIRST

The searching algorithm is applied now in its alternative form to produce a function which will search the tree in a depth-first fashion. Remember that the queue which controls the search is now updated by adding extra nodes to the head of the queue. The function is coded as follows.

```
(DEFUN DEPTH-FIRST NIL
 (TERPRI)
 (FORMAT T "Type starting node.")
 (TERPRI)
 (SETF L (LIST (READ)))
 (DO ((X 1 (+ 1 X)))
  ((NULL L))
  (SETF HEAD (FIRST L))
  (TERPRI)
  (FORMAT T "Node visited No.")
  (PRIN1 X)
  (FORMAT T " is ")
  (PRIN1 HEAD)
  (TERPRI)
  (IF (AND (GET HEAD 'CHILD) (NULL (GET HEAD
   'VISITED))) (EXTEND-QUEUE) (SETF L
    (REST L)))
   (FORMAT T "Current queue is ")
   (PRIN1 L)))
```

First, the starting node is requested by `TERPRI` and `FORMAT T` and assigned to `L` by `(SETF L (LIST (READ)))`. The DO loop then repeats until the queue `L` has no more nodes.

For convenience the first node in the queue is bound to the identifier HEAD by the expression (SETF HEAD (FIRST L)). FORMAT T and PRIN1 are then used to show this current node being examined.

The essential part of the algorithm is then coded into the conditional expression which follows. If the node at the head of the queue has children, these are added to the front of the queue, provided that the node has not already been examined. This is an essential extra condition to include because repeated nodes are inevitable in the search. If the condition were not present, then an endless return to the same nodes would take place. The two conditions have to be linked by AND. The first condition is true if (GET HEAD 'CHILD) is not NIL, which naturally is the case when child nodes are linked to the node considered. The second condition is true when the node has not been examined before, as determined by the indicator 'VISITED.

If the double condition evaluates as NIL, the queue is simply shortened to remove the current node. This is achieved by the expression (SETF L (REST L)). Alternatively, if the condition evaluates as T an additional function, EXTEND-QUEUE is used to add the fresh nodes to the head of the queue.

```
(DEFUN EXTEND-QUEUE NIL
 (SETF L (APPEND (GET HEAD 'CHILD) L))
 (PUTPROP HEAD T 'VISITED))
```

First, SETF and APPEND are employed in placing the new nodes, obtained by (GET HEAD 'CHILD) at the head of the head of the queue L. After this the fact that the node has been checked is recorded by (PUTPROP HEAD T 'VISITED).

The use of the function in searching the same example of a tree structure is shown here.

```
* (DEPTH-FIRST)
Node visited No.1 is A
Current queue is (B C D A)
Node visited No.2 is B
Current queue is ( E F B C D A)
Node visited No.3 is E
Current queue is (H I J E F B C D A)
Node visited No.4 is H
Current queue is (I J E F B C D A)
Node visited No.5 is I
Current queue is (J E F B C D A)
Node visited No.6 is J
Current queue is (E F B C D A)
Node visited No.7 is E
Current queue is (F B C D A)
Node visited No.8 is F
Current queue is (B C D A)
```

```
Node visited No.9 is B
Current queue is (C D A)
Node visited No.10 is C
Current queue is (G C D A)
Node visited No.11 is G
Current queue is (K L G C D A)
Node visited No.12 is K
Current queue is (L G C D A)
Node visited No.13 is L
Current queue is (G C D A)
Node visited No.14 is G
Current queue is (C D A)
Node visited No.15 is C
Current queue is (D A)
Node visited No.16 is D
Current queue is (A)
Node visited No.17 is A
Current queue is NIL
NIL
```

The Single-server Queue

The elementary concept of queueing which has been introduced in the algorithms for depth-first and breadth-first search techniques above can now be developed in more detail. Intuitive ideas on what occurs in a queue must be formalised before two illustrations of queues can be constructed as LISP functions. These two examples, `WAITING` and `CHECK-OUT`, are the first introduction to simulation techniques. Simulations will then be treated in more detail in Chapters Seven and Eight.

As has already been stated, the main structure of a queueing system consists of three parts. These are the arrival mechanism, the queueing discipline and the service mechanism. Differences in any of these will lead to different queue structures. For the first simulation program, `WAITING`, a very simple queue structure is specified. The second simulation, `CHECK-OUT`, will be more sophisticated.

In `WAITING`, the arrival mechanism is wholly deterministic, rather than a statistical, stochastic process. The user is allowed to choose how many arrivals there will be in the queue and also the time of each of these arrivals. Clearly, this is not realistic for many situations, although it could reflect a very well ordered appointments system. The queueing discipline in `WAITING` makes three principal assumptions. The first in, first out rule is applied. No exits are permitted from the queue, except via the service mechanism. There is no theoretical limit to the length of the queue which can arise. The service mechanism in the simulation allows only one server. `WAITING` is thus an example of the 'single-server' queue. The service time for each 'customer' is also presumed to be the same. The simulation therefore allows not only all the arrival times to be specified in advance, but also all the times at which customers are served.

When a simulation of any kind is written, it must be decided exactly what information is required from the program and how these results are to be presented. With a queue simulation, for example, the total number of customers in a given period might be wanted, or their mean waiting time, or how long the servicing mechanism is lying idle. In this first simulation, the length of the queue at specified intervals is obtained and presented to the user in tabular form. However, before the first of the functions contained in the simulation is developed, it is best to consider an example of how the separate arrivals and servicing events result in the generation of a queue in practice. In this illustration, the development of a queue over a period of 50 seconds is deduced from the arrival and service times. Arrivals occur randomly at 6, 7, 16, 23, 31, 33, 44 and 48 seconds. Service times are more regular at 13, 26 and 39 seconds. The length of the queue is measured at five second intervals. A diagram can be used in which the letter a represents arrivals and the letter s represents the service mechanism reducing the queue. The five second intervals are also shown.

```
0    5   10   15   20   25   30   35   40   45   50
l----l----l----l----l----l----l----l----l----l----l
       a a       a     a         a a          a    a
l----l----l----l----l----l----l----l----l----l----l----l
            s              s          s
l----l----l----l----l----l----l----l----l----l----l----l
   0    2    1    2    3    2    4    3    4    5
```

The bottom row of numbers give a simple deduction of the queue length for the relevant time interval. Each value is found by subtracting the number of servicing events from the number of arrivals and adding the queue length carried over from the previous interval. This is the basic algorithm used in the simulation. It will be coded in the function BUILD-QUEUE. Before this, the initial data for the queue have to be entered and the first function involved in this is CONDITIONS.

Selecting the Initial Constraints - CONDITIONS

Here, the duration of the simulation, the length of the intervals over which the queue length is to be calculated and the time between serving events can all be entered.

```
(DEFUN CONDITIONS NIL
 (TERPRI)
 (FORMAT T "Type duration of simulation.")
 (TERPRI)
 (SETF DURATION (READ))
 (FORMAT T "seconds.")
 (TERPRI)
 (FORMAT T "Type display period.")
 (TERPRI)
 (SETF PERIOD (READ))
 (FORMAT T "seconds.")
```

```
(TERPRI)
(FORMAT T "Type service time.")
(TERPRI)
(SETF SERVICE-TIME (READ))
(FORMAT T "seconds."))
```

First, the function requests the length of the simulation exercise and binds the user's entry to the identifier DURATION with the expression (SETF DURATION (READ)). An extra FORMAT T adds the unit 'seconds' to the number typed. TERPRI then forces a fresh line on the display. Analogous code follows to acquire values for PERIOD, which is the interval length, and SERVICE-TIME, which gives the frequency of servicing for the queue.

A function which is associated with CONDITIONS is RESET.

User Fallibility and Error Trapping - RESET

Often extra features have to be included in a program to prevent the person operating it from doing something foolish. It is necessary to make a crash of the program as difficult as possible. There are some very obvious traps to avoid. For example, division by nought can be made impossible if any, potentially zero, entry is guarded against at the input stage. The programming technique which tries to guess what user-errors may threaten the integrity of the program being constructed is called error trapping. It is shown here for the program WAITING with the design of the additional function RESET. This results from the fact that the function CONDITIONS has requested the user to enter the duration of the simulation exercise and also the time intervals over which the queue length is to be measured. The duration should be a multiple of this interval, but the user may not have realised this. Values could have been typed in for which the duration is not a multiple of the interval. RESET will correct such an error. It is coded as follows.

```
(DEFUN RESET NIL
 (SETF A (/ DURATION PERIOD))
 (SETF B (INTEGER A))
 (SETF STEPS (IF (> A B) (1+ B) B))
 (SETF DURATION (* STEPS PERIOD)))
```

In order to appreciate the operation of the function it is easier if values are assigned to DURATION and PERIOD. Suppose the former is 27 seconds and the latter 5 seconds. Here, the length of the simulation period is not a multiple of the time interval and a correction needs to be made. The first expression in the function, (SETF A (/ DURATION PERIOD)), will give the variable A the value 5.4 and then (SETF B (INTEGER A)) makes B equal to 5. The number of intervals for the simulation is calculated as the value of STEPS. The conditional expression makes STEPS equal to B only when A is not greater than B and thus the duration is a multiple of the interval. When the duration is not a multiple of the interval, STEPS becomes (1+ B). With the figures given, STEPS will be calculated as 6. Finally, the value of DURATION itself is

readjusted to the appropriate value by the last expression in RESET, (SETF DURATION (* STEPS PERIOD)).

The operation of RESET in calculating the number of time intervals and adjusting the size of DURATION can be seen in the following example, which uses the values already assumed above. At this point the value entered for the service time is not significant but is included here to indicate the remaining display created by CONDITIONS.

```
* (CONDITIONS)
Type duration of simulation.
27 seconds.
Type display period.
5 seconds.
Type service time.
3 seconds.
NIL
* DURATION
27
* (RESET)
30
* STEPS
6
```

This example shows how RESET has automatically returned the new value required for DURATION, as well as calculating the appropriate number of intervals as STEPS. It is perhaps worth mentioning again the distinction between RESET, which is a function and so requires placing within parenthesis, and STEPS, which is an ide tifier and is evaluated immediately without the need for brackets.

Selecting Arrival Times - JOIN-QUEUE

A function must now be written which allows the arrival times to be entered into WAITING. It is coded like this.

```
(DEFUN JOIN-QUEUE NIL
 (TERPRI)
 (FORMAT T "Type number of arrivals.")
 (TERPRI)
 (SETF ARRIVALS (READ))
 (SETF A-TIME (MAKE-ARRAY ARRIVALS))
 (DOTIMES (X ARRIVALS T)
          (TERPRI)
          (FORMAT T "Type Arrival time No.")
          (PRIN1 (1+ X))
          (TERPRI)
          (SETF (AREF A-TIME X) (READ))
          (FORMAT T "seconds.")))
```

First, the total number of arrivals to the queue over the period being analysed is requested. It is assigned to the variable ARRIVALS by (SETF ARRIVALS (READ)). After this an array is established with (SETF A-TIME (MAKE-ARRAY ARRIVALS)). This places just the right number of locations into A-TIME to store all the arrival times. These are requested during the DOTIMES loop. As in the function CONDITIONS, the screen display is arranged neatly with the use of TERPRI, FORMAT T and PRIN1.

At this point, all the information needed to create the queue simulation has been obtained, but before the principal function BUILD-QUEUE can be constructed, two new Common LISP primitive will be explained.

Common LISP Primitives - ZEROP, >=

ZEROP is a predicate function which returns T if its argument evaluates as zero and NIL otherwise. Here are examples of its use.

```
* (ZEROP (- 24 (* 4 6)))
T
```

The expression given as the argument for ZEROP becomes, in conventional arithmetical notation, (24 - 24), or zero, and so ZEROP causes the whole expression to evaluate as T.

```
* (ZEROP (+ 3 9))
NIL
```

In this expression, the argument for ZEROP is 12, so the expression yields NIL.

The 'less than or equals' predicate function returns T or NIL in precisely the way which would be expected from conventional mathematical usage.

```
* (<= 6 12)
T
* (<= 7 7)
T
* (<= 9 2)
NIL
```

Exactly analogous is the 'greater than or equals' predicate, which is not used in the following function.

Constructing the Queue - BUILD-QUEUE

Now the queue itself can be established. In this simulation, all the necessary information is available at the beginning. The simulation is completely deterministic

and so the status of the queue at any of the intervals to be considered can be calculated in advance. This is quite unlike the second simulation to be developed below, CHECK-OUT, which develops in a statistical way. The queue length at the end of each interval is calculated by first adding the number of fresh arrivals to the previous length. The number of servicing events which take place during the interval is then subtracted. This gives the current number in the queue. Obviously, if the rate at which the queue is being shortened by the server is greater than the rate at which it is being increased by arrivals, this simple algorithm could lead to a negative queue length. Potential negative values are therefore replaced by zero. The algorithm is coded at the end of the function because before it can be applied, two arrays, S-TIME and QUEUE have to be created.

```
(DEFUN BUILD-QUEUE NIL
 (SETF SERVICES (INTEGER
  (/ DURATION SERVICE-TIME)))
(SETF S-TIME (MAKE-ARRAY SERVICES))
(DOTIMES (X SERVICES T)
         (SETF (AREF S-TIME X)
          (* (1+ X) SERVICE-TIME)))
 (SETF QUEUE (MAKE-ARRAY STEPS))
 (DOTIMES (X STEPS T)
          (SETF J 0)
          (SETF L 0)
          (SETF TA (* X PERIOD))
          (SETF TB (+ TA PERIOD))
          (DOTIMES (Y ARRIVALS T)
                   (IF (AND (> (AREF A-TIME Y)
                    TA) (<= (AREF A-TIME Y) TB))
                     (SETF J (1+ J)) T))
 (DOTIMES (Y SERVICES T)
          (IF (AND (> (AREF S-TIME Y) TA)
           (<= (AREF S-TIME Y) TB))
            (SETF L (1+ L)) T ))
 (SETF Q (IF (ZEROP X) (- J L) (+ (- J L)
  (AREF QUEUE (1- X)))))
 (SETF (AREF QUEUE X) (IF (> Q 0) Q 0))))
```

First, all the servicing events must be placed into an array. The total number of such events is found by dividing the duration of the simulation by the service time. Clearly, fractional events are meaningless, and so the integral value of the calculation is required. The result is assigned to the variable SERVICES by the expression (SETF SERVICES (INTEGER (/ DURATION SERVICE-TIME))). Its value can then be used to establish the array S-TIME with (SETF S-TIME (MAKE-ARRAY SERVICES)). It is this array which holds all the times of the servicing events during the simulation. A DOTIMES loop is used to calculate each entry for S-TIME. The array entries are inevitably a multiple of the service time. However, as the variable X has the initial value of zero, it must be incremented by one before being used to

multiply SERVICE-TIME. This leads to the expression used in the DOTIMES loop, (SETF (AREF S-TIME X) (* (1+ X) SERVICE-TIME)).

Next an array, QUEUE, is established to record the lengths of the queue at the end of each interval. As the total number of intervals is STEPS, this value is used to set up the array. The rest of the function's code is then contained within one DOTIMES loop which uses the value of X over the range of STEPS to consider each interval in succession, deducing the total increase or decrease in length. At the beginning of the loop, four variables are introduced which represent the state of any of the individual queues. (SETF J 0) and (SETF L 0) are expressions giving initial values of zero to J, the number of arrivals to the queue, and L, the number leaving the queue during that particular interval. (SETF TA (* X PERIOD)) and (SETF TB (+ TA PERIOD)) set up the times at the beginning and end of each interval.

The first Y loop nested within the main DOTIMES loop increments J for any arrivals to the queue during the interval defined by the value of X. All the locations in the A-TIME array are examined in turn, as a result of the range specified for Y when the loop is set up. Each location is checked against the value of TA and TB for the interval. The conditional embedded in the loop uses AND to combine two expressions, (> (AREF A-TIME Y) TA) and its complementary, (<= (AREF A-TIME Y) TB). Both of these are satisfied when the particular A-TIME value involved lies within the interval. This means that an arrival occurs in the queue during at this time. Because both component expressions then evaluate as T, the AND does as well and the first action of the IF is triggered. This is (SETF J (1+ J)) and so the the number of arrivals recorded for the interval is incremented appropriately. Because the Y loop investigates all the A-TIME locations, any arrival during the period will be registered.

In a very similar process, the second nested Y loop checks through all the service events recorded at the beginning of BUILD-QUEUE in the S-TIME array. Analogous conditions compare the S-TIME locations with the beginning and end of the interval, TA and TB, and the variable L is incremented if a service event takes place during the interval.

After both Y loops have been completed, the values of J and L are used to establish the net length of the queue for the interval. This new value is assigned to the variable Q. A complication is introduced by the fact that the length of the queue in the previous interval must also be added to the current total. There is no difficultly in referring to this value for most of the intervals. For the majority of them, the previous interval's queue length will be stored in the QUEUE array and accessed by (AREF QUEUE (1-X)). However, this previous location does not exist for (AREF QUEUE 0), at the beginning of the array and any accidental reference made to it will cause an error. To avoid this, the expression calculating the value of Q has to employ more stringent conditions. The new predicate function, ZEROP, is used. If (ZEROP X) evaluates as T, then the first interval is being considered and the queue length is simply (- J L). Here, the number of service events is subtracted from the number of arrivals. For all

other X values, the length of the previous queue must be included. This gives (+ (- J L) (AREF QUEUE (1-X))) as the new queue length.
Provision is now made for the possibility that Q will be assigned a negative value. Negative Q is meaningless. The queue length stored in the QUEUE array must only be equal to Q if it is positive. If the number of servicing events has exceeded the number of arrivals, the queue length must be zero because obviously servicing events cannot be projected into the future. This is achieved by (SETF (AREF QUEUE X) (IF (> Q 0) Q 0)).

With the completion of BUILD-QUEUE, the essential part of the simulation has been achieved. All that needs to be done to complete the program WAITING is to write a function which prints out the results.

Displaying the Queue Length - PRINT-QUEUE

All the information processed by the simulation now resides in the QUEUE array. PRINT-OUT presents this information on the screen and is coded as follows.

```
(DEFUN PRINT-QUEUE NIL
 (DOTIMES (X STEPS T)
          (TERPRI)
          (FORMAT T "Queue length for step ")
          (PRIN1 (1+ X))
          (FORMAT T " is ")
          (PRIN1 (AREF QUEUE X))))
```

The DOTIMES loop explicitly uses the value of STEPS, which is the total number of intervals considered, to control the number of times it repeats. TERPRI and FORMAT T then state which queue length is being printed with the message 'Queue length for step', followed by (PRIN1 (1+ X)). After this, the value of the QUEUE array location for this interval is added to the display.

Simulating a Single-server Queue - WAITING

All the component functions are now arranged in the correct order to give the complete simulation.

```
(DEFUN WAITING NIL
 (CONDITIONS)
 (RESET)
 (JOIN-QUEUE)
 (BUILD-QUEUE)
 (PRINT-QUEUE))
```

A typical execution of the program is shown here:

```
* (WAITING)
Type duration of simulation.
47 seconds.
Type display period.
5 seconds.
Type service time.
13 seconds.
Type number of arrivals.
8
Type arrival time No.1
6 seconds.
Type arrival time No.2
7 seconds.
Type arrival time No.3
16 seconds.
Type arrival time No.4
23 seconds.
Type arrival time No.5
31 seconds.
Type arrival time No.6
33 seconds.
Type arrival time No.7
44 seconds.
Type arrival time No.8
48 seconds.
Queue length for step No.1 is 0
Queue length for step No.2 is 2
Queue length for step No.3 is 1
Queue length for step No.4 is 2
Queue length for step No.5 is 3
Queue length for step No.6 is 2
Queue length for step No.7 is 4
Queue length for step No.8 is 3
Queue length for step No.9 is 4
Queue length for step No.10 is 5
```

The simulation can be used to experiment with the effect of different service and arrival times on the growth of the queue. It is not necessary to run the entire program again. If, for example, the user wished to repeat the simulation with the same set of arrival times but a different service time, the relevant functions could be called directly. SETF is used to reassign the new value for SERVICE-TIME. Then BUILD-QUEUE constructs the new queue lengths before they are shown by PRINT-QUEUE. The effect of a more frequent service upon the queue can be illustrated.

```
* (SETF SERVICE-TIME 4)
4
* (BUILD-QUEUE)
T
* (PRINT-QUEUE)
Queue length for step No.1 is 0
Queue length for step No.2 is 1
Queue length for step No.3 is 0
Queue length for step No.4 is 0
Queue length for step No.5 is 0
Queue length for step No.6 is 0
Queue length for step No.7 is 1
Queue length for step No.8 is 0
Queue length for step No.9 is 0
Queue length for step No.10 is 0
```

Alternatively, the increase in the queue length when the service is infrequent can also be modelled by assigning a far larger value to SERVICE-TIME.

```
* (SETF SERVICE-TIME 49)
49
* (BUILD-QUEUE)
T
* (PRINT-QUEUE)
Queue length for step No.1 is 0
Queue length for step No.2 is 2
Queue length for step No.3 is 2
Queue length for step No.4 is 3
Queue length for step No.5 is 4
Queue length for step No.6 is 4
Queue length for step No.7 is 6
Queue length for step No.8 is 6
Queue length for step No.9 is 7
Queue length for step No.10 is 7
```

The simulation of a queue generated by the program WAITING can be made more realistic by replacing some of the functions, while keeping others the same. The second queue simulation which is developed, CHECK-OUT, retains CONDITIONS and RESET as in WAITING, and uses the same function, BUILD-QUEUE, to calculate the queue lengths in different intervals. However, it differs from WAITING in two important respects. CHECK-OUT provides a more detailed screen display by representing the queue length and service events in a diagramatic fashion. More significantly, the simulation does not utilise JOIN-QUEUE to record fixed arrival times. Instead a statistical function, STOCHASTIC, will be written to simulate a typical distribution of additions to the queue, given a mean arrival time. However, before the necessary code

can be developed, another function resident in Golden Common LISP must first be investigated.

Common LISP Primitive - GET DECODED TIME

The internal clock of the IBM PC-compatible is accessed by use of this instruction. As the name implies, the time is given in a readable fashion by the function, displayed in the order second, minute, hour, date, month and year.

```
* (GET-DECODED-TIME)
38
28
11
29
11
1987
```

The usefulness of the function for present requirements is that it can provide a 'seed' value in the generation of random numbers. Algorithms employed to give a random number usually require some starting value which is itself not too obvious. The internal clock creates such initial vagueness in a convenient way. For example, an unpredictable value could be assigned to the atom A by this expression.

```
* (SETF A (GET-DECODED-TIME))
36
```

The principle will be developed further in the following sections. Before this, though, the use of GET-DECODED-TIME can be illustrated in a short timing routine which could be employed for various purposes.

Telling the Time - CLOCK

CLOCK displays the number of seconds which have elapsed since the last time the function was called. In its present form, it assumes that the previous evaluation has taken place within the preceding minute, but the function could easily be modified to extend its use.

```
(DEFUN CLOCK NIL
 (SETF NOW (GET-DECODED-TIME))
 (SETF DELAY (+ DELAY (IF (> NOW THEN)
  (- NOW THEN) (+ (- 60 THEN) NOW))))
 (SETF THEN NOW)
 (TERPRI)
 (FORMAT T "Elapsed time is ")
 (PRIN1 DELAY)
 (FORMAT T " seconds."))
```

Before CLOCK is called for the first time, the initial values of the variables DELAY and THEN are initialised, independently of the function, as zero and the current value of (GET-DECODED-TIME) respectively. As soon as CLOCK is evaluated, the current value of the function GET-DECODED-TIME is assigned to the variable NOW. If this value is greater than that stored as THEN, the elapsed time is simply (- NOW THEN). Alternatively, if the value of NOW is less than that of THEN, the seconds figure returned by GET-DECODED-TIME has passed 60 and commenced again as zero. In this case, the elapsed time will be given by the expression (+ (- 60 THEN) NOW). These two possibilities are combined in the fresh evaluation of DELAY, (+ DELAY (IF (> NOW THEN) (- NOW THEN) (+ (- 60 THEN) NOW))). The value is then printed on the screen with accompanying text. CLOCK will operate like this.

```
* (SETF DELAY 0)
0
* (SETF THEN (GET-DECODED TIME))
13
* (CLOCK)
Elapsed time is 10 seconds.
NIL
* (CLOCK)
Elapsed time is 25 seconds.
NIL
```

The function could be simplified to give just a numerical value to be used in any other function which required timing, for example, a 'real time' simulation.

Pseudo-random Numbers

With the new function GET-DECODED-TIME, an attempt can be made to construct a function which will provide random numbers. These are frequently required in a simulation exercise because they give the element of chance which is always present in the real world. Unfortunately, a basic contradiction is involved in writing a deterministic function which, nevertheless, feigns randomness. The alternative solution would be to store a large set of suitably chosen random numbers, initially obtained from some external source, within the program. Memory requirements prevent this from being really feasible, so an algorithm which produces sufficiently unpredictable numbers must be written. The best which can be hoped for is the generation of a series of pseudo-random numbers.

Various methods have been devised which simulate such quasi-randomness. One is called the congruence method. Here, three numbers, A, B, C are selected together with an initial 'seed' value as the initial random number. A new random number is then calculated by this equation.

$$RANDOM = (A * SEED + B) \text{ MOD } C$$

If A, B and C had the values 3, 4 and 5, and if a seed value of 9 were chosen, the new random number returned would be calculated as (3 * 9 + 4) MOD 5 or 31 MOD 5. This

is 1. Now 1 is used as a fresh seed value, yielding (3 * 1 + 4) MOD 5 or 2. Small numbers like these do not lead to a long series of apparently chance values, but with much larger and suitably chosen initial coefficients in the equation reasonable randomness can be obtained.

Another standard algorithm for deriving pseudo-random numbers is the mid-square method. Suppose the seed value is now the four digit number 3721. It is squared to give 13 845 841. The middle four digits are taken as a new random number, 8458. Squaring this in turn provides 5377 as the middle digits of 71 537 764. Numbers obtained in this fashion can be divided by 10 000 to give pseudo- random decimals in the range zero to one, which is the conventional size expected for random numbers in programming. This mid-square algorithm, which was devised by one of the founding-figures in computing science, John von Neumann, is used as the basis of the next function.

A Practical Random Number Function - RANDOM

RANDOM is a short function which will produce pseudo-random two figure decimal numbers, R, in the range 0 < R < 1. It uses the mid-square method, beginning with an initial seed value which needs to be obtained separately before the function is called. For example, if the seed value is 43, the function will calculate the squared number, 1849. The middle two digits are then selected as the next seed value, 84, and the function returns 0.84 as the current pseudo-random number. The steps involved in this process are best considered as a flowchart before any attempt is made at coding into LISP (see Figure 6.4).

The flowchart in Figure 6.4 highlights the following stages. First, the seed is selected, here 43. It is then squared to give 1849. The number is divided by 1000, yielding 1.849. After this the integral part is subtracted to leave 0.849. Now the number is multiplied by 100, producing 84.9. The new seed is the integral value of this, which is 84. If at this point the seed were less than 31, it would be doubled until it was greater. This is to ensure that at the next call to RANDOM the squared value of the new seed has the four digits assumed by the algorithm. Finally, the seed is divided by 100 to produce the decimal pseudo-random number required. All of these individual stages are coded like this.

```
(DEFUN RANDOM NIL
 (SETF A (/ (SQUARE SEED) 1D3))
 (SETF SEED (INTEGER (* (- A
  (INTEGER A)) 1D2)))
 (DO ((X 1 (1+ X)))
     ((> SEED 31))
     (SETF SEED (* SEED 2)))
 (/ SEED 1D2))
```

It can be seen that several stages of the flowchart are condensed into single expressions. Thus the fourth, fifth and sixth operations in the flowchart are all present in the expression (SETF SEED (INTEGER (* (- A (INTEGER A)) 1D2))). The function will now give results like these.

```
* (SETF SEED 43)
43
* (RANDOM)
0.84D0
* (RANDOM)
0.4D0
```

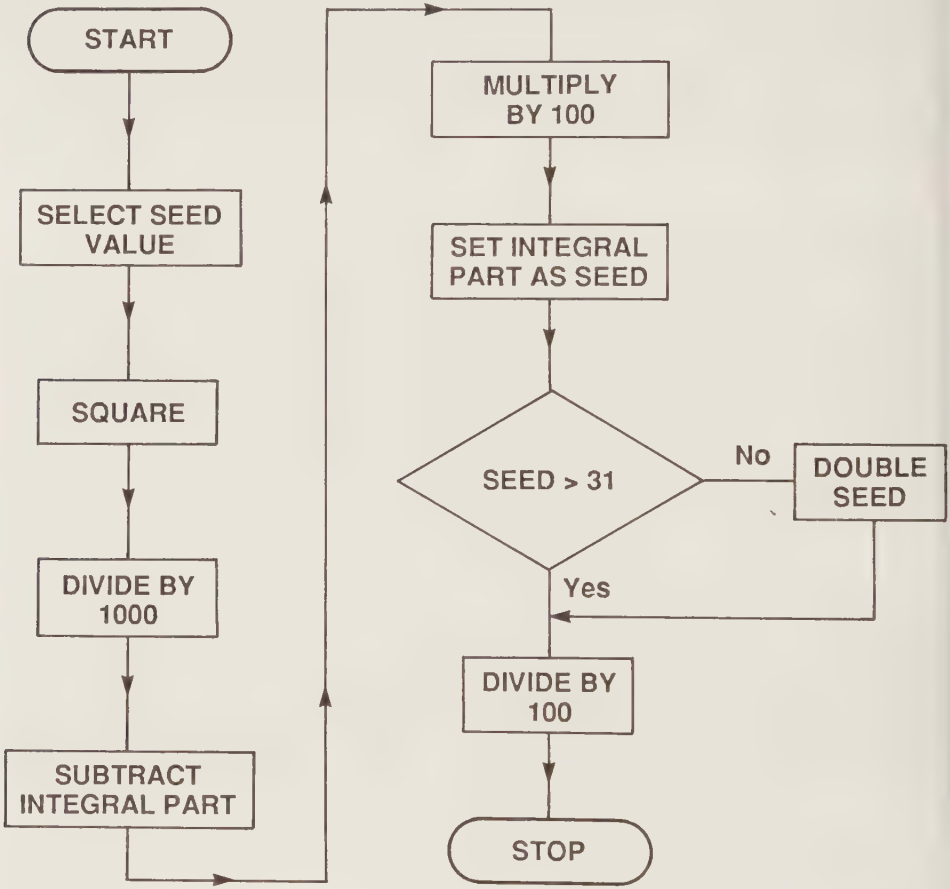


Figure 6.4

Testing Randomness - RAND, TEST-RAND

The efficiency of RANDOM as a suitable number generator needs to be checked before it can be used in the second queueing simulation, CHECK-OUT. In addition, a method for choosing the initial value of SEED, employed by RANDOM, is required. This will be made easier if the typical output of RANDOM for different values of SEED is examined.

To help illustrate this, the function RAND prints the next eight numbers generated by RANDOM when SEED is assigned the number N as its starting point.

```
(DEFUN RAND (N)
 (SETF SEED N)
 (DOTIMES (X 8 T)
          (PRIN1 (RANDOM))
          (FORMAT T " ")))
```

After the variable SEED is assigned N as the first seed value, a DOTIMES loop calculates eight random numbers and prints them on a single line of display. FORMAT T is used to separate the individual numbers neatly. Thus a typical result could be found by taking 52 as the first seed.

```
* (RAND 52)
0.7D0 0.89D0 0.92D0 0.46D0 0.44D0 0.93D0 0.64D0 0.36D0
```

Rather than constantly using RAND with different arguments, the function can be embedded in a loop to print automatically the first eight numbers generated from a sequence of seed values. This is carried out in the function TEST-RAND.

```
(DEFUN TEST-RAND (N)
 (DO ((X N (1+ X)))
     ((> X (+ 10 N)))
     (PRINT X)
     (TERPRI)
     (RAND X)))
```

The DO loop now calculates the random numbers over a range of different seeds, beginning with the one typed as the argument for TEST-RAND.

```
* (TEST-RAND 52)
52
0.7D0 0.89D0 0.92D0 0.46D0 0.44D0 0.93D0 0.64D0 0.36D0
53
0.8D0 0.39D0 0.52D0 0.7D0 0.89D0 0.92D0 0.46D0 0.44D0
54
0.91D0 0.56D0 0.52D0 0.7D0 0.89D0 0.92D0 0.46D00.44D0
55
0.32D0 0.32D0 0.32D0 0.32D0 0.32D0 0.32D0 0.32D0 0.32D0
56
0.52D0 0.7D0 0.89D0 0.92D0 0.46D0 0.44D0 0.93D0 0.64D0
57
0.48D0 0.6D0 0.59D0 0.48D0 0.6D0 0.59D0 0.48D0 0.6D0
58
0.36D0 0.58D0 0.36D0 0.58D0 0.36D0 0.58D0 0.36D0 0.58D0
```

```
59
0.48D0 0.6D0 0.59D0 0.48D0 0.6D0 0.59D0 0.48D0 0.6D0
60
0.59D0 0.48D0 0.6D0 0.59D0 0.48D0 0.6D0 0.59D0 0.48D0
61
0.72D0 0.36D00 0.58D0 0.36D0 0.58D0 0.36D0 0.58D0 0.36D0
62
0.84D0 0.4D0 0.59D0 0.48D0 0.6D0 0.59D0 0.48D0 0.6D0
```

Selecting the Seed Value - RANDOMISE

Examination of the numbers produced by TEST-RAND will show how different seeds vary in their usefulness in generating convincing randomness. Fortunately, an extra degree of chance can be employed by using GET-DECODED-TIME, as suggested above, to provide the first value. This is done in the function RANDOMISE.

```
(DEFUN RANDOMISE NIL
 (SETF SEED (GET-DECODED-TIME))
 (DO ((X 1 (1+ X)))
     ((> SEED 40))
     (SETF SEED (* SEED 2)))
 SEED)
```

Here, SEED is assigned the value of GET-DECODED-TIME. The loop then eliminates seeds that are too small. The number 40 is an arbitrary choice, though a reasonable one. RANDOMISE would then yield seed values in this fashion.

```
* (RANDOMISE)
47
```

RANDOM is not the only additional function required before a statistical distribution of arrivals to the queue can be simulated. In order to create a realistic pattern of arrivals, it is necessary to be able to calculate natural logarithms. This is done by the next function to be written.

Calculating Natural Logs - LOGARITHM

LOGARITHM uses a standard expansion to calculate the natural logarithm of a number. This is the Taylor series which can be found in any standard text on mathematics.

$$\log N = (N-1) - 1/2\,(N-1)^2 + 1/3\,(N-1)^3 - 1/4\,(N-1)^4 \ldots$$

Writing the LISP function which calculates this is made simpler if the term (N-1) is replaced by the equivalent quantity -(1-N). The new series can be coded quite readily because all the terms now have the same sign.

$$\log N = -(1-N) - 1/2\,(1-N)^2 - 1/3\,(1-N)^3 - 1/4\,(1-N)^4 \ldots$$

LOGARITHM is written as follows.

```
(DEFUN LOGARITHM (N)
 (SETF A (1- N))
 (SETF B (- 1 N))
 (DO ((X 2 (1+ X)))
     ((< B 1D-10))
     (SETF B (* B (- 1 N)))
     (SETF A (- A (/ B X)))) A )
```

The value of log N is calculated with the use of two variables, A and B. A is the running total of the terms of the expansion. B is used to help calculate each of them in turn. It can be seen that A is first assigned the quantity given at the beginning of the series, `-(1- N)`, by `(SETF A (1- N))`. The only surprise here might be the clearly illustrated divergence of LISP's arithmetic from conventional notation. B is given the initial value `(- 1 N)`. This is multiplied by itself at each pass through the DO loop and, combined with the value of the loop variable, is just the correct quantity to include in the running total of A with the expression `(SETF A (- A (/ B X)))`. The execution of the loop continues, with the calculation of ever diminishing extra terms, until the EXIT condition of `(< B 1D-10)` is met. The value of the logarithm is then returned by the final A in the function.

Evaluation of an expression like `(LOGARITHM 0.41)` will yield the appropriate values.

Simulating Arrival Times - STOCHASTIC

The new functions now available, RANDOMISE, LOGARITHM and RANDOM, can now be utilised in writing the function which generates a random and statistically plausible, series of queue arrivals. This is STOCHASTIC and is coded as shown below.

```
(DEFUN STOCHASTIC NIL
 (RANDOMISE)
 (SETF A-TIME (MAKE-ARRAY 20))
 (SETF ELAPSED 0)
 (DO ((X 0 (1+ X)))
     ((> ELAPSED DURATION))
     (SETF ARRIVALS (1+ X))
     (SETF ELAPSED (+ ELAPSED
      (- (* MEAN-TIME (LOGARITHM (RANDOM))))))
     (SETF (AREF A-TIME X) ELAPSED)))
```

First, RANDOMISE selects the initial seed value that will be needed to begin generating random numbers. The array A-TIME is set up after this with an arbitrarily chosen number of locations. Twenty is assumed to be sufficient to store all the arrival times of a typical simulation exercise, but the number could quite readily be adjusted if it was found in practice to be too small. Use is made of the variable ELAPSED in the

function. It measures the total amount of time which would pass while the various queue arrivals were gathering. Of course, this is a purely theoretical value, as the simulation does not operate in real time. Nevertheless, ELAPSED is essential to the algorithm which selects random arrivals. It is initialised by (SETF ELAPSED 0).

The arrivals are then calculated in the DO loop. This repeats until ELAPSED has exceeded the total length of the simulation, as recorded in DURATION. This does, in fact, lead to the last arrival being redundant in that it lies outside the time allocated, but this does not cause an error in the operation of the program. At each pass through the loop, (SETF ARRIVALS (1+ X)) sets the value of the variable ARRIVALS to the total number of times the loop has been executed. This is a convenient way of counting the total number of arrivals to the queue. For the simpler simulation, WAITING, this value was, of course, typed in by the user in the function JOIN-QUEUE.

Vital part of the function lies in the next symbolic expression, (SETF ELAPSED (+ ELAPSED (- (* MEAN-TIME (LOGARITHM (RANDOM)))))). This multiplies the mean time of arrivals, MEAN-TIME, which will be chosen by the user in CHECK-OUT, by a random logarithmic expression which imitates the statistical arrival of new members of the queue. The minus sign is used to make the time increment positive rather than the negative quantity otherwise given. It is then added to the existing value of ELAPSED. The value of ELAPSED is placed into the arrivals array A-TIME at each pass through the loop by (SETF (AREF A-TIME X) ELAPSED). The array thus holds a series of arrival times which increase steadily over the duration of the simulation and which are used later in BUILD-QUEUE.

Exits from the Queue - SERVER-STATUS

One of the differences between WAITING and CHECK-OUT, the statistical simulation of queue arrivals, has now been achieved with the new function, STOCHASTIC. The other modification mentioned above was a more graphic representation of the appearance of the queue on the monitor. In order to show exactly what the queue will look like at the end of each interval, an extra function is required which will show whether the server is attending to a fresh member of the queue at this point. The function is called SERVER-STATUS. It examines the times of the servicing events which are placed into the S-TIME array by the function BUILD-QUEUE. These are compared with the lengths of the intervals to calculate which intervals contain one or more servicing event, or none at all. These intervals are then represented on the screen display, in the later function SHOW-QUEUE, by a different diagram representing the server than the one used for those intervals in which no servicing event occurs.

```
(DEFUN SERVER-STATUS NIL
 (SETF SERVER (MAKE-ARRAY STEPS))
 (DOTIMES (X STEPS T)
          (SETF TA (* X PERIOD))
          (SETF TB (+ TA PERIOD))
          (DOTIMES (Y SERVICES T)
                   (IF (AND (> (AREF S-TIME Y)
                    TA) (<= (AREF S-TIME Y) TB))
                     (SETF (AREF SERVER X)
                      (1+ (AREF SERVER X))) T))))
```

A new array is required to represent the involvement of the server in the queue discipline. There is already the S-TIME array, but this simply records the actual times of the servicing events. It does not make any reference to what is taking place in each individual interval. The new array records whether a particular interval contains a servicing event or not. The number of locations in the array hence needs to be equal to the number of intervals in the simulations's display. This is the value of the variable STEPS. The new array is therefore established by (SETF SERVER (MAKE-ARRAY STEPS)).

After this, two nested DOTIMES loops place appropriate values into the locations of the array SERVER. The outer loop considers each interval in turn. TA and TB are calculated, as in the function BUILD-QUEUE, as the time at the beginning and end of each interval. The Y loop then checks all of the service times in the S-TIME array to discover if any lie between TA and TB. If they do, the newly-established array is updated by the expression (IF (AND (> (AREF S-TIME Y) TA) (<= (AREF S-TIME Y) TB)) (SETF (AREF SERVER X) (1+ (AREF SERVER X))) T). The process is very similar to the creation of the QUEUE array in BUILD-QUEUE. At the end of both loops the array will consist of a series of numbers, zero representing intervals in which no servicing event takes place and non-zero values indicating intervals where the queue is being attended. The SERVER array values are then used in the function which pictures the queue on the monitor, SHOW-QUEUE.

Waiting Your Turn - SHOW-QUEUE, BLANK

Here, the queue is created as a diagram on the screen.

```
(DEFUN SHOW-QUEUE (Q S)
 (SPACE)
 (BLANK 5)
 (FORMAT T "QUEUE")
 (BLANK 14)
 (FORMAT T "SERVER")
 (TERPRI)
 (BLANK 24)
 (FORMAT T "------")
 (TERPRI)
 (BLANK 5)
 (DOTIMES (Y Q T)
          (FORMAT T "X"))
 (BLANK (- 21 Q))
 (IF (ZEROP S) (BLANK 5) T)
 (FORMAT T "X")
 (TERPRI)
 (BLANK 24)
 (FORMAT T "------"))
```

SHOW-QUEUE uses the values obtained from the QUEUE and the SERVER arrays. These are represented as Q and S on the parameter list. After this, the function. SPACE, consisting simply of (TERPRI) using twice, first places an empty line on the screen. Then a new function, BLANK, indents the next part of the display. This is just a DOTIMES loop which uses FORMAT T a specified number of times to create a gap.

```
(DEFUN BLANK (B)
 (DOTIMES (X B T)
          (FORMAT T " ")))
```

The two parts of the display are shown as 'QUEUE' and 'SERVER', separated by (BLANK 14). Then TERPRI and BLANK are used again to allow the latter heading to be underlined by (FORMAT T "------"). Next, (BLANK 5) again indents the display, followed by a DOTIMES loop which uses the value of Q to print a series of capital letters representing the current queue length.

The length of the gap which comes next on the display will depend upon the length of the queue. To ensure that the server part of the display always appears in the same column of the display, a variable gap is created, which is shortened for longer queues, by (BLANK (- 21 Q)). The server display will then vary according to whether a service event occurs in the current interval. This will be determined by the value of S passed to the function and obtained from the SERVER array. ZEROP is employed in the expression (IF (ZEROP S) (BLANK 5) T). If S is zero, this expression results in extra spaces being inserted before a capital letter X is printed by (FORMAT T "X"). This indicates that a customer has just left the queue and that no current serving event is taking place. However, if S is not zero, and a service event does fall within the interval, the letter X is placed beneath the 'R' of 'SERVER'. This shows that the queue is being served.

Finally, TERPRI, BLANK and FORMAT are used again to underline the server position on the screen.

A Pictorial Queue Simulation- CHECK-OUT

The extra functions can now be added to those written for the earlier simulation to create the new program, CHECK-OUT.

```
(DEFUN CHECK-OUT NIL
 (CONDITIONS)
 (RESET)
 (TERPRI)
 (FORMAT T "Type mean arrival time.")
 (TERPRI)
 (SETF MEAN-TIME (READ))
```

```
 (FORMAT T "seconds.")
 (STOCHASTIC)
 (BUILD-QUEUE)
 (SERVER-STATUS)
 (DOTIMES (X STEPS T)
          (SPACE)
          (PAUSE)
          (FORMAT T "Queue at step No.")
          (PRIN1 (1+ X))
          (SHOW-QUEUE (AREF QUEUE X)
           (AREF SERVER X)))
(SPACE)
(FORMAT T "Number of service events: ")
(PRIN1 SERVICES)
(TERPRI)
(FORMAT T "Number of arrivals: ")
(PRIN1 (1- ARRIVALS))
(DOTIMES (X (1- ARRIVALS) T)
         (TERPRI)
         (FORMAT T "Arrival No.")
         (PRIN1 (1+ X))
         (FORMAT T " at ")
         (PRIN1 (INTEGER (AREF A-TIME X)))
         (FORMAT T " seconds."))
(TERPRI)
(FORMAT T "Queue simulation ended."))
```

First, CONDITIONS and RESET obtain necessary information to begin the simulation. Next, FORMAT T, TERPRI, SETF and READ create the section of the function needed to obtain a value for MEAN-TIME. This is required in STOCHASTIC, which fills the A-TIME array. After this, BUILD-QUEUE determines the value for the QUEUE array and SERVER-STATUS fills the locations in SERVER. A DOTIMES loop follows which inserts the values of QUEUE and SERVER into the display function, SHOW-QUEUE, so that the queue can be displayed. PAUSE is also used in the loop to make the display easier to read.

When CHECK-OUT is executed, this loop shows the queue structure for each of the intervals. At the end, the remaining part of the function prints out the total number of service events, the number of the arrivals and each arrival time. INTEGER is needed to suppress unwanted decimal accuracy in the last set of values.

CHECK-OUT will now operate as illustrated in this typical display of its results.

```
* (CHECK-OUT)
Type duration of simulation.
98 seconds.
Type display period.
15 seconds.
Type service time.
23 seconds.
Type mean arrival time.
10 seconds.
Queue at step No.1

          QUEUE                SERVER
                               ------
          X                          X
                               ------

Queue at step No.2

          QUEUE                SERVER
                               ------
          XX                     X

                               ------

Queue at step No.3

          QUEUE                SERVER
                               ------
          XXXX                       X
                               ------

Queue at step No.4

          QUEUE                SERVER
                               ------
          XXXXX                  X
                               ------

Queue at step No.5

          QUEUE                SERVER
                               ------
          XXXXXX                 X
                               ------
```

```
Queue at step No.6

          QUEUE                SERVER
                               ------
          XXXXXXXX                   X
                               ------

Queue at step No.7

          QUEUE                SERVER
                               ------
          XXXXXXXXX              X
                               ------

Number of service events: 4
Number of arrivals: 13
Arrival No.1 at 10 seconds.
Arrival No.2 at 15 seconds.
Arrival No.3 at 25 seconds.
Arrival No.4 at 31 seconds.
Arrival No.5 at 41 seconds.
Arrival No.6 at 46 seconds.
Arrival No.7 at 57 seconds.
Arrival No.8 at 62 seconds.
Arrival No.9 at 72 seconds.
Arrival No.10 at 78 seconds.
Arrival No.11 at 88 seconds.
Arrival No.12 at 93 seconds.
Queue simulation ended.
```

Recursion in LISP

A powerful structure available in LISP is recursion. This permits a function to call itself and results in far more concise coding. It can be a difficult concept to grasp at first, as it can seem rather like cheating. You decide what your function is going to be able to do, assume it can actually do whatever is required and then employ this fact in the initial function definition. It is best to approach the concept via a simple example. Suppose you have decided to write a function which will take a list as its argument and return upon evaluation the number of elements which it contains. Thus, if you present COUNT with a five element list, you want it to evaluate as 5, like this:

```
* (COUNT '(BIKE CAR LORRY PLANE SHIP))
5
```

Remember that at this stage the function is quite imaginary. You have not written it, but you can begin in the normal way, with DEFUN and the parameter list containing the list L:

```
(DEFUN COUNT (L)
 (Body of function))
```

You then make the vital mental leap and realise that if L has no elements at all, (COUNT L) must return zero. On the other hand, if L is not empty, (COUNT L) must return a value one greater than the value which it would for a list with one element fewer. This, of course, is given by (COUNT (REST L)). You can regard this schematically:

L contains no elements: (COUNT L) returns NIL

L does contain elements: (COUNT L) returns (1+ (COUNT (REST L)))

This calls quite obviously for the COND structure, leading to this function definition:

```
(DEFUN COUNT (L)
 (COND ((NULL L) 0)
       (T (1+ (COUNT (REST L))))))
```

Applying this to a list shows that the function works.

```
* (COUNT '(WINTER SPRING SUMMER AUTUMN))
4
```

It will seem very much like magic at first and perhaps remind you of the famous Escher print in which two hands are busy drawing one another, in a horribly convincing way. However, it does work. Recursive functions are particularly economical and invariably far more compact than equivalent iterative definitions.

Considering the above example more closely will show that the essential feature of recursion which has allowed the function definition to work is twofold. Firstly, the ((NULL L) 0) clause has allowed the function an escape condition and prevented it from endlessly calling itself. Secondly, the (T (1+ (COUNT (REST L)))) clause has ensured that the recursive function call is altering each time in a way which eventually will inevitably trigger the escape. Successful recursive definitions will require that these two features are included.

A recursive definition does not have to be restricted to a two part COND expression. For example, the following definition for a function, PART-OF, is more complicated. PART-OF takes an element and a list for its arguments and returns T if the element is contained in the list and NIL otherwise. It is coded like this.

```
(DEFUN PART-OF (E L)
 (COND ((NULL L) NIL)
       ((EQUAL E (FIRST L)) T)
       (T (PART-OF E (REST L)))))
```

If the list L is empty, E cannot be a member of it and so the COND expression evaluates as NIL. However, if E is the same as the first element of the list, (EQUAL E (FIRST L)) will evaluate as T and the second clause of the COND will return T. Other cases are checked by the recursive call to PART-OF in the final clause of the COND. Inevitably, if E is contained in L, it will eventually be the same as the first element of the constantly depleted list. The COND will then evaluate as T. Otherwise (NULL L) will finally evaluate as T and the COND return NIL. Here, the function is seen in operation.

```
* (PART-OF 'A '(A B C D))
T

* (PART-OF 'Z '(A B C D))
NIL

* (SETF FRANKENSTEIN '(HEAD BRAIN HEART
LIVER))
(HEAD BRAIN HEART LIVER)

* (PART-OF 'LIVER FRANKENSTEIN)
T

* (PART-OF 'SOUL FRANKENSTEIN)
NIL
```

Summary

1. LISP is ideal for creating tree data structures.

2. A breadth-first or depth-first search can be coded quite easily using a queue to control the order in which nodes are checked.

3. The queue data structure can be simulated in LISP to model actual queuing in the real world. This can be of either single-server or multi-server queues.

4. NOT is a LISP primitive which reverses the Boolean value of its argument.

5. Error trapping attempts to protect a function against failure caused by a careless user.

6. ZEROP returns T if its argument evaluates as zero and NIL otherwise.

7. GET-DECODED-TIME provides access to the internal clock.

8. GET-DECODED-TIME can give seed values for random number algorithms.

Exercise 6

State how the following evaluate.

1. `(ZEROP (- (* 3 4) (* 6 2)))`

2. `(ZEROP (* (+ 4 7) (- 2 9)))`

3. `(NOT (ZEROP (- 20 (* 4 5))))`

7

Life-like LISP: Biological Simulation

An important and quite unique role of the computer is the extremely easy way in which it allows various mathematical, physical or other processes to be copied, or simulated. There are many things which would be rather difficult to achieve using the real world as the laboratory but which can, instead, be recreated on a pleasantly manageable scale in a computational model. Such simulations have rapidly provided a major use for computers and range over a broad field of application. Children in schools can make authentic investigations of Egyptian tombs or experience the excitement of searching for a wrecked galleon. At a greater level of sophistication pilots test their reactions in a variety of circumstances without risking their lives or an expensive aircraft. Pure scientific research is aided by simulation techniques when, for example, astronomers are able to test cosmological models of the early universe and study their evolution in an infinitesimal fraction of the time which the simulated process would have taken had it really occurred. More trivial applications of simulations also exist in profusion, as witnessed by the electronic arcade games, which inevitably become closer to their chosen reality with each successive summer season on the pier.

In theory, there is no reason why simulations could not be carried without the aid of the computer. Indeed, they have been. The great advantage of electronic computing, though, is the way in which very large numbers of calculations can be performed quickly. This allows the interrelation of different factors in the simulation to be decided quickly enough to produce a numerical model of the real system in a sufficiently short period of time to allow reasonable conclusions to be drawn about the system's behaviour. Admittedly, in some circumstances even the largest computers have difficulty in producing convincing results. This can occur when an extremely large number of factors are involved. For example, modelling weather patterns in the atmosphere is still difficult. Similarly, the 'nuclear winter' theory of Carl Sagan and others was later challenged on the grounds that insufficient factors had been included in the mathematical model of the Earth's atmosphere.

System Dynamics and Simulation

One approach to writing a computer simulation involves systems dynamics. This is a method in which a particular set of related causes and effects are isolated as much as possible from the external environment to form a closed system, the investigation of which is far easier to conduct. The precise definition of what the system is going to

consist may be quite difficult and in some cases apparently arbitrary. Nevertheless, if a simulation is going to be modelled in a computer program, simplifications must be carried out. There is little chance of including all possible features of the real situation and so care must be taken in selecting the most dominant.

In doing this, causal loops should be considered. A causal loop results when a number of factors influence one another, eventually reaching the beginning of the same chain again. For example, raising the rent on a property may cause resentment on the part of the tenant. This might reveal itself in the form of neglecting the fabric of the building, failing to report water leaks, not bothering to clear weeds from the path, spilling sump oil on the drive and other deliberate negligences. These in turn will lead to a deterioration of the house and an increase in the cost of maintaining it. The landlord might then be obliged to raise the rent again... The whole process is like the traditional music hall song 'There's a hole in my bucket'. It is an example of positive feedback, or a 'reinforcing' causal loop.

Negative feedback also exists, in which a 'compensating' causal loop tends to offset the factors creating a particular situation. The more pussy sharpens his claws on the Chippendale, the more often a copy of *Country Life* bounces off his back, in a way which disenchants him with period furniture.

Examples of causal loops are found in the biological simulation developed in this chapter.

Screen Display in Simulations

Before the first simulation program is developed, more attention must be given again to the presentation of information on the monitor. Quite obviously there is little point in devising intricate algorithms, which allow the simulation of a complicated process, if the results are displayed in a confusing way to the user. Investigated here is `EXPLODE`, a function which can help to enhance a screen display. However, first the general application of this function in LISP software needs to be outlined.

The Spoken Word

Many applications of LISP lie in the exciting world of Artificial Intelligence. An especially important objective in this discipline is the possibility of the eventual construction of software which can deal reasonably efficiently with natural language. One reason for this is that operators need to be able to communicate with their machines in as relaxed a way as possible, typing in instructions which are close to the language which they would use in ordinary conversation. Even more intriguing are developments being made in voice control of robotic machinery. Undoubtedly, when natural language communication with computers and robots is possible human society will acquire a new dimension, as well as interesting new philosophical problems.

However, true understanding of natural language is still well beyond the reach of existing computers and programming languages. It is significant that Terry Winograd, who made a significant breakthrough by restricting man/machine communication to a limited 'microworld' in his seminal work *'Understanding Natural Language'* (1972), still accepts that there is a long way to go. His later *'Language as a Cognitive Process'*, he admits, will be inevitably only the first of a series of texts cataloguing the developments gradually being made in language processing.

Nevertheless, it is remarkable just how easy it is to write software which mimics comprehension of speech in a superficial, though initially convincing, manner. The notorious example of this is Joseph Weizenbaum's ELIZA, which employed a clever set of linguistically orientated algorithms to guide its communication with a human partner into apparently cogent discourse. The program relied upon a complicated series of purely automatic responses to the text being entered.

Splitting an Atom

The success of ELIZA depended upon the way in which seemingly subtle nuances in human language frequently depend upon minor differences in syntactic form. For example, the two sentences in this minimalist horror story:

```
JAN WAS AGHAST. IAN WAS A GHOST!
```

have distinct meanings, though the actual differences between then, viewed in terms of the physical symbols that have been typed, are relatively few.

It is possible to construct relatively simple functions which can detect typographical differences like these. The important LISP primitive which helps to analyse the structure of individual words, and which therefore allows slight changes to be distinguished, is `EXPLODE`. This permits an atom to be broken down into its individual characters. These are returned as a list. Thus `(EXPLODE 'ENIWETOK)` gives `(E N I W E T O K)`. This function, however, was not available in the version of Golden Common LISP used in the compilation of this book. Developing the function from the available primitives will require the introduction of three new Golden Common LISP functions, `MAPCAR`, `STRING` and `COERCE`, as well as the principle embodied in 'lambda expressions'.

Common LISP Primitive - MAPCAR

The function `MAPCAR` provides another method by which iteration can be carried out in LISP. A simple example, of some triviality, will be outlined. Suppose it is necessary to multiply three pairs of numbers together in succession:

3 x 5 = 15
7 x 2 = 14
8 x 6 = 48

The numbers could be entered and then multiplied in a DOTIMES loop.

```
* (DOTIMES (X N T)
(PRINT (* (READ) (READ))))3
5
15
7
2
14
8
6
48
```

Alternatively, the MAPCAR function could operate on both sets of three numbers, entered as lists like this. Note the hash sign and the quote before the function itself.

```
* (MAPCAR #'* '(3 7 8) '(5 2 6))
(15 14 48)
```

MAPCAR has taken the multiplicative function with arguments in the form of lists. It has then calculated each individual product for respective pairs of elements in turn and presented its results in the form of another list.

Lambda Expressions - LAMBDA

A lambda expression gives the equivalent of a function definition without having to use DEFUN. The first element in the expression is the atom LAMBDA. Then follows a list of arguments. After this are the s-expressions which constitute the definition. A lambda expression is very convenient in association with MAPCAR and can be used in this way to operate on each of the elements of a list.

Common LISP Primitives - COERCE, STRING

COERCE will take a LISP item and convert it to a different form. For example, a string might be coerced to a list. It takes two arguments, the item itself followed by the type it is to be coerced into. The Golden Common LISP manual should be consulted to see what possibilities are available and also for the object types which can be returned as a string by the use of the function STRING.

Resolving a Symbol - EXPLODE, IMPLODE

These functions will now be used in the creation of the function EXPLODE. This also requires the setting up of an array, LETTER, which will hold the letters of the alphabet for use in EXPLODE. LETTER is initialised and filled by the function ABC.

```
(DEFUN ABC NIL
 (SETF LETTER (MAKE-ARRAY 26))
 (DOTIMES (X 26 T)
          (TERPRI)
          (FORMAT T "LETTER ")
          (PRIN1 (1+ X))
          (FORMAT " ")
          (SETF (AREF LETTER X) (READ))))
```

This function simply uses MAKE-ARRAY to establish LETTER and then the DOTIMES loop to request the letters of the alphabet in this fashion.

```
* (ABC)
LETTER 1 A
LETTER 2 B
LETTER 3 C
.........

LETTER 26 Z
```

The function EXPLODE is now coded like this.

```
(DEFUN EXPLODE (SYMBOL)
 (MAPCAR #'(LAMBDA (A) (AREF LETTER (- A 65)))
  (COERCE (STRING SYMBOL) 'LIST)))
```

Its argument, SYMBOL, is coerced to a numerical list by the expression (COERCE (STRING SYMBOL) 'LIST). A lambda expression, operating within MAPCAR, then returns the individual letters of the original argument using the values in the LETTER array.

The function can be seen operating here.

```
* (EXPLODE 'SUPERNOVA)
(S U P E R N O V A)
```

The inverse function to EXPLODE can also be written. It is IMPLODE, which will take a list and compact it to a string, in this fashion:

```
* (IMPLODE '(P U L S A R))
"PULSAR"
```

It does not operate quite like the IMPLODE available in some dialects of LISP, but is still useful and will be employed in a later program. It is coded as two functions, IMPLODE itself and ASC.

```
(DEFUN IMPLODE (L)
 (COERCE (MAPCAR #'ASC L) 'STRING))

(DEFUN ASC (A)
 (FIRST (COERCE (STRING A) 'LIST)))
```

Simulating Cell Growth - BACTERIA

The first simulation program to be written is called BACTERIA. It closely resembles the simulation game of LIFE devised at Cambridge University in 1970. This became very popular amongst programmers on the main-frame computers of the day and almost notorious for its efficiency in squandering valuable machine time. This caused some commercial organisations to ban it completely. Later, with the advent of the domestic micro-computer in the late 1970s, it acquired a second group of addicts. Versions of the simulation then appeared written in the machine-specific BASIC of Commodore PETs, Sinclair ZX81s and other early home micro-computers. The program's popularity is partly the result of its combining a relatively simple algorithm with an effective and plausible representation of the real world.

The simulation is of cell growth in the culture medium of a petri dish. The dish is shown as a series of rows of full stops on the monitor. Each of these sites can develop into a cell, displayed as a capital letter O. At the beginning of the simulation, the user decides which sites are to be seeded with a bacterial cell. After this, the program decides how such an initial distribution will develop. The colony of microbes may perish. The dish may become completely infected. A static population might result, with a dynamic equilibrium between the birth of new cells and the death of older. Or more interesting things may happen, as shown in the display of an actual program execution included later in this chapter.

Simulation Time-step

An important principle in the simulation is that the cell population is considered in terms of cycles of growth and degeneration. During any particular cycle, cells will be born and others die, depending upon the distribution of cells at the beginning of the cycle. Then, at the end of the cycle, the new distribution of cells will be used to determine what growth and extinction occurs in the next. The overall evolution of the population can be regarded as taking place in discrete stages, with the initial conditions presented to one cycle being the final state of the cycle preceding it.

This division into specific stages reflects periods of elapsed time in the real world. If the bacterial culture were actually living, the processes of birth and death would take place in finite time periods, represented within the simulation by the stages already described. The period of elapsed time in the real world is referred to as the simulation

time step, as opposed to the computational time required to simulate the equivalent evolution of the mathematical model assumed in the program. The concept of time-step is important in simulation techniques. In some simulations, perhaps the modelling of a financial problem, the computational time will be considerably shorter than the simulation time-step. In economic problems, the latter may be several years compared with only hours of machine time. Alternatively, the situation could be reversed. In weather forecasting, developments taking only a few hours in the atmosphere may require far longer periods on even the largest machines. In the important category of real-time simulations, the simulation time-step and the computational time will be identical. The flight simulators used for training shuttle astronauts are an example of this.

The discrete cycles of bacterial development are reflected in the program BACTERIA by the use of two separate representations of the cell population, the lists CELLS and COPIES. The former determines the cell distribution at the beginning of each cycle. The latter shows the new distribution which evolves during the course of the cycle. At the end of each time-step, the contents of COPIES has to be transferred to CELLS before the next cycle can be executed.

BACTERIA's Principal Algorithm

The algorithm used consists of two parts. One governs the generation of cells and the other their extinction. Each site on the grid representing the culture dish has eight neighbours, those above and below and to the left and right, and the four additional diagonally related sites. If any empty site in the dish has exactly three neighbouring sites which are occupied with cells, a fresh cell is created to fill the vacancy. This simulates the natural reproduction of a bacterial culture. Naturally, some regions of cell growth may subsequently die, likely causes being isolation or, conversely, over-population. The program BACTERIA simulates this aspect of the evolution of the colony of cells by specifying that any cell with fewer than two, or more than three, occupied neighbouring sites will be eliminated.

An example will show how this simple, two part algorithm does create change. Three initial cells will be considered, identified here on a grid rather like that of the game 'battleships'. This means that the three cells have been placed on the sites D3, E3 and F3.

```
5 . . . . . .
4 . . . . . .
3 . . . O O O . . .
2 . . . . . .
1 . . . . . .
  A B C D E F G H I
```

In order to decide whether any new cell growth will take place, the sites bordering the region of existing cells must be examined in turn to discover whether any of them have the required three neighbours. For example, site C4 has just one neighbouring cell, at D3. When all the sites have been checked, there are two which do have three neighbouring cells. These are the sites E4 and E2. New cells will be created at these locations. Next, the existing cells must be considered to see if they will survive into the second cycle. As cells D3 and F3 each possess only one neighbouring cell, they will perish. Cell E3, which has two neighbours, does survive. This means that at the end of the first cycle there are still three cells on the grid, but at the locations E2, E3 and E4.

```
5 . . . . . .
4 . . . . O . . . .
3 . . . . O . . . .
2 . . . . O . . . .
1 . . . . . .
  A B C D E F G H I
```

The second cycle now begins. Following exactly the same method as before it will be discovered that fresh cells are created at D3 and F3. The cells at E4 and E2 die. Cell E3 survives. The grid now again becomes like this.

```
5 . . . . . .
4 . . . . . .
3 . . . O O O . . .
2 . . . . . .
1 . . . . . .
  A B C D E F G H I
```

Quite clearly, the initial pattern of cells selected has led to a stable, though oscillating, population.

Carefully working through a simple application of the algorithms selected for a simulation is an important exercise. It forces the programmer to think at a very basic level about the operation of the algorithms and to discover any hidden anomalies. Perhaps more essentially, it provides immediate material which can be tested against the results produced by early versions of the program. This helps with debugging.

This algorithm will now be coded into LISP to create the simulation. A number of functions are required. DISH sets up the mathematical model of the culture dish and establishes the fundamental methods of knowledge representation embedded in BACTERIA. DISPLAY-CELLS creates the screen display showing the user the evolution of the colony of cells entered. The function SEED allows this initial colony to be selected. NEIGHBOURS determines the number of occupied sites around any particular site on the dish. CYCLE decides what evolution of cells should take place in each time-step. Finally, the overall function BACTERIA combines the various parts of the program to give an easily used simulation with a clear screen display.

Before DISH, the first function embodied in BACTERIA, can be written, another LISP primitive function must be introduced. This is SET.

Common LISP Primitive - SET

The function SETF has been used so frequently that its operation should be well understood. There is another LISP primitive, SET, which can also be used to assign values, but which has a subtle difference. This can be demonstrated by typing in two functions, TESTA and TESTB, which are identical except that in the second a SETF is replaced by a SET.

```
(DEFUN TESTA NIL
 (SETF A (READ))
 (SETF A '(X Y Z)))

(DEFUN TESTB NIL
 (SETF A (READ))
 (SET A '(X Y Z)))
```

Neither function performs a very exciting role, but the way in which each behaves should make clear the distinction between the two primitives. The intention is that the value of a symbol typed in by the user should be changed to (X Y Z). Suppose that L has been previously assigned the value (P Q R). Which of the two functions will alter this can be tested as follows.

```
* (TESTA) L
(X Y Z)
* L
(P Q R)

* (TESTB) L
(X Y Z)
* L
(X Y Z)
```

It can be seen that only TESTB has altered the value of L. This is because the primitive, SET, employed in the second test function evaluates both of its arguments before the assignment is made. SETF, used exclusively until now, has only evaluated its second argument. Hence TESTA simply assigned (X Y Z) to A, but TESTB assigned (X Y Z) after A had been evaluated to the symbol L, the value of A resulting from the previous expression in the function (SETF A (READ)).

The use of SET will become more clear after it is utilised in the first of BACTERIA's component functions, DISH.

A LISP Petri Dish - DISH

It is now possible to construct the first function required by BACTERIA. This is DISH, the function which sets up the mathematical culture dish in which the cells of the simulated microbes will be able to thrive.

As anticipated above, the culture dish will be represented by ten rows of 20 dots upon the monitor screen. Each living cell is shown as a capital letter O, while the spaces between cells are marked by full stops. This is a smaller area of display than used in the original versions of the simulation, but is sufficient, nevertheless, to illustrate its mode of operation. A typical display will occupy a section of the left-hand side of the display and could appear like this.

```
ROW A ....................
ROW B ....................
ROW C ....................
ROW D .....OOOO...........
ROW E ....................
ROW F ....................
ROW G ....................
ROW H ................O...
ROW I ....................
ROW J ....................
```

In order to aid the user's identification of particular cell sites, the ten rows are labelled ROW A, ROW B ... ROWJ. The first of the cells shown here will therefore be the sixth in ROW D. The isolated cell is the 17th in ROW H. The cells are recorded internally in various functions of BACTERIA in almost exactly the same way. Each row of cells is associated with a list of the same name, except that the space is omitted between the word ROW and the identifying letter. Additionally, the user will regard the extreme left hand cell in each row as being the first, although in the coding the same cell will be numbered zero, in accordance with the use of the LISP primitive NTH.

The cells or empty spaces of each row are recorded very directly in each list as T or NIL. This means that the ROW D shown above will be the LISP list ROWD consisting of these 20 elements:

```
(NIL NIL NIL NIL NIL T T T T NIL NIL NIL NIL NIL NIL NIL
 NIL NIL NIL NIL)
```

Here is the function DISH:

```
(DEFUN DISH NIL
 (SETF CELLS '(ROWA ROWB ROWC ROWD ROWE ROWF ROWG
 ROWH ROWI ROWJ))
 (SETF COPIES '(ROWK ROWL ROWM ROWN ROWO ROWP
  ROWQ ROWR ROWS ROWT))
 (DOTIMES (X 9 T)
          (PUTPROP (NTH X CELLS)
           (NTH (1+ X) CELLS) 'AFTER)
          (PUTPROP (NTH (1+ X) CELLS)
           (NTH X CELLS) 'BEFORE))
 (SETF ROWA NIL)
 (DOTIMES (X 20 T)
          (SETF ROWA (CONS NIL ROWA)))
 (MAPCAR #'(LAMBDA (ROW) (SET ROW ROWA)) (APPEND
  CELLS COPIES))
 (MAPCAR #'(LAMBDA (ROW1 ROW2) (PUTPROP ROW1 ROW2
  'COPY)) CELLS COPIES))
```

The lists representing the rows of cells are first introduced as the ten elements of an overall list, CELLS, which therefore records the contents of the entire dish. There are two reasons for doing this. Placing each list into CELLS makes it easier to identify the two rows either side of any of the rows in the dish with the use of property lists. Also, using MAPCAR with CELLS allows the initial contents of each row to be established in a convenient fashion.

A duplicate dish is created in addition to CELLS. Such a copy is required by some of the functions which subsequently simulate the growth of the cells in their culture medium. The 200 cells of this duplicate dish are all recorded by the ten elements of the overall list, COPIES. This list is set up in an analogous way to CELLS.

The next step is to choose an easy way of recording the adjacent rows to any of the ten in the dish. This is done by the DOTIMES loop. Note that this loop needs to repeat only nine times, although there are ten lists being considered. At each pass through the loop, PUTPROP is used to place on to the property list of the first nine lists of CELLS the row which follows each of them. This is achieved using the indicator AFTER in the s-expression (PUTPROP (NTH X CELLS) (NTH (1+ X) CELLS) 'AFTER). In a similar way, the preceding row of each of the last nine lists of CELLS is placed on to the property lists by the expression (PUTPROP (NTH (1+ X) CELLS) (NTH X CELLS) 'BEFORE). After this, the two rows adjacent to any row of the dish can be addressed by an appropriate use of GET with one of the indicators AFTER or BEFORE. For example, (GET 'ROWD 'BEFORE) evaluates as ROWC, and (GET 'ROWF 'AFTER) has the value ROWG.

DISH next sets up ROWA as a 20 element list consisting simply of repeated NILs. First, (SETF ROWA NIL) binds the identifier ROWA to NIL. Then the DOTIMES loop which follows is able to build up ROWA to the required length by the repeated use of the symbolic-expression (SETF ROWA (CONS NIL ROWA)).

Once just one of the lists, here ROWA, has been established, all of the others can be set up very directly, simply by using it as the basic model to copy. This is done using MAPCAR with a suitable lambda expression to operate on all of the 20 lists, presented as the combined list (APPEND CELLS COPIES). There is an unnecessary duplication of ROWA, but this is not too much of a problem. The lambda expression (LAMBDA (ROW) (SET ROW ROWA)) uses SET to evaluate each of the lists in CELLS and COPIES and assign the 20 NILS which represent one of the rows of empty sites in the model of the dish.

Finally, MAPCAR is used a second time to place each of the duplicate rows in COPIES on to the property list of the corresponding row in CELLS. The lambda expression needed this time is (LAMBDA (ROW1 ROW2) (PUTPROP ROW1 ROW2 'COPY)). ROW1 takes each list in turn from CELLS. ROW2 selects the lists of COPIES. Then the use of PUTPROP places the duplicate list on to the appropriate property list, with COPY as the identifier.

In summary, DISH performs the following role in BACTERIA. It sets up the initial representation of the culture dish as ten lists each consisting of 20 elements. Each of the corresponding rows is related to the preceding and following rows via its property list. Additionally, the property lists determine the duplicate copy of each row in a second representation of the dish. This second version of the dish is needed to record the evolution of the cell population in each time-step of the simulation.

Three further LISP primitives are required before the function can be written which modifies a list from CELLS. These are BUTLAST and NTHCDR, which select elements from the head and tail of another list respectively, and the function EVAL.

Common LISP Primitive - BUTLAST

This function returns a list minus a specified number of the elements at its end.

```
* (BUTLAST '(A B C D E F) 4)
(A B)
```

Naturally, the original list is left intact unless it is deliberately altered by using SETF, as this example will illustrate:

```
* (SETF L '(A B C D E F))
(A B C D E F)
* (BUTLAST L 4)
(A B)
* L
(A B C D E F)
* (SETF L (BUTLAST L 4))
(A B)
* L
(A B)
```

Common LISP Primitive - NTHCDR

This primitive function returns the tail of a list, beginning at an element selected by the user.

```
* (NTHCDR 4 '(A B C D E F))
(E F)
```

Note that again the elements of the list have been counted from zero.

Common LISP Primitive - EVAL

EVAL allows an extra stage of evaluation to take place for a symbolic expression, as will be seen happening in the function developed subsequently.

Seeding the Dish with Microbes - ALTER, SEED

When the user wants to plant a bacterial cell in one of the rows, the name of the row and the number of the site, assuming that site No.1 is the first on the extreme left, are typed in. BUTLAST and NTHCDR are then used to assemble a new list in which the selected element is changed from NIL to T. It is easier to understand how this is achieved if a shorter list of only seven elements is considered. Suppose that the list which represents a row of seven empty sites is initially:

```
(NIL NIL NIL NIL NIL NIL NIL)
```

If the fifth site is to be seeded with a cell, the list must become:

```
(NIL NIL NIL NIL T NIL NIL)
```

This can be done by taking the head and tail of the existing list, omitting the fifth element, and recombining them with an enclosed T. The new list will consist of the three parts:

```
(NIL NIL NIL NIL)  T  (NIL NIL)
```

Using BUTLAST and NTHCDR to determine the head and tail, and calling the list ROWX, the three parts will become:

```
(BUTLAST ROWX (- 8 5))  T  (NTHCDR 5)
```

It is important to see that although the numerical argument given to NTHCDR is simply the number of the selected site, the argument required by BUTLAST is the difference between the site number and one more than the length of the row. This modification will also be appropriate for the 20 element lists in CELLS.

Identical reasoning produces the function ALTER. This takes ROW and N as its arguments, directly corresponding to the ROWX and 5 of the simpler example given. However, it also has the argument VAL. This is designed to take either the value T or

NIL as the replacement element in the new list. This extension of the routine allows ALTER to be used later in CYCLE, when it can replace expiring cells with a value of NIL in the appropriate row.

```
(DEFUN ALTER (ROW N VAL)
 (SET ROW (APPEND (BUTLAST (EVAL ROW) (- 21 N))
  (CONS VAL (NTHCDR N (EVAL ROW))))))
```

EVAL has to be used to give an extra level of evaluation of ROW. This is because BUTLAST and NTHCDR need to operate on the full list of 20 elements and not simply an identifier, like ROWA. APPEND and CONS then permit the complete new list to be assembled from the three component sublists.

ALTER is now used by the function SEED.

```
(DEFUN SEED NIL
 (TERPRI)
 (FORMAT T "Type row, then position.")
 (TERPRI)
 (ALTER (READ) (READ) T))
```

FORMAT T places a suitable prompt on the screen. READ then allows the user to select the row in which the new cell is to be added and the position of the chosen site within the row. The value given for VAL in ALTER is, naturally, T because this marks the presence of a cell as opposed to an empty site.

The next function required by BACTERIA uses a new LISP primitive.

Common LISP Primitive - LAST

LAST returns the final element of a list as a single element list.

```
* (LAST '(X Y Z))
(Z)
```

If the result is needed as an atom, FIRST can be combined with LAST.

```
* (FIRST (LAST '(X Y Z)))
Z
```

The Contents of the Dish - DISPLAY-CELLS

This function shows the contents of all the lists contained within CELLS as the 10 X 20 grid of full stops and the capital letter O:

```
(DEFUN DISPLAY-CELLS NIL
 (SETF ROW 'ROWA)
 (DO ((X 1 (1+ X)))
     ((NULL ROW))
     (TERPRI)
     (FORMAT T "ROW ")
     (PRIN1 (FIRST (LAST (EXPLODE ROW))))
     (FORMAT T " ")
     (DOTIMES (Y 20 T)
              (IF (NTH Y (EVAL ROW))
               (FORMAT T "O") (FORMAT T "."))))
 (SETF ROW (GET ROW 'AFTER))))
```

The identifier ROW is assigned the value of the first of the lists in CELLS with (SETF ROW ROWA). In the DO loop which follows, the name of the row is placed on the left-hand side of the screen. For clarity, a gap is introduced between the word 'ROW' and its letter suffix. TERPRI and FORMAT T display the word itself, while the function EXPLODE is used to determine the final letter in the identifier. All the characters are obtained as a list by (EXPLODE ROW) and the last letter is printed by the expression (PRIN1 (FIRST (LAST (EXPLODE ROW)))).

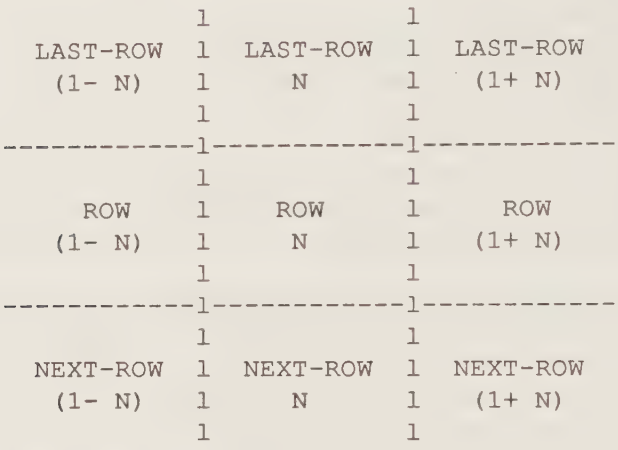
After a space has been added by (FORMAT T " "), a DOTIMES loop checks through the elements in the current list. In order to allow this to happen, (EVAL ROW) is employed to evaluate ROWA as its whole list of 20 elements. The conditional expression contained in the loop will then use (FORMAT T "O") to print a capital O on the screen if the relevant element identified by (NTH Y (EVAL ROW)) has the value T. If instead its value is NIL, (FORMAT T ".") will mark the empty site with a full-stop.

At the end of the DOTIMES loop, but still within the overall DO loop, the identifier, ROW, is assigned to the next list in CELLS by (SETF ROW (GET ROW 'AFTER)). Note that this time EVAL is not used to evaluate ROW further. This is because the property list is linked to the symbol rather than to its value. When the final row is reached, (GET ROW 'AFTER) returns NIL as the value of ROW. The condition ((NULL ROW)) at the beginning of the DO loop is then satisfied and the function ends. At this point, all 200 cells and sites have been placed on the screen, as well as the name of each row.

Testing the Environment - NEIGHBOURS

A crucial test in BACTERIA is checking the immediate environment of each site to count the number of neighbouring cells. Then an application can be made of the rule that extinction results from there being fewer than two, or more than three, other cells in the vicinity. Additionally, all empty sites need to be checked in order to find those with at least three adjacent cells. Here, a fresh cell will be created.

A diagram will help to define the eight neighbours of the site identified by the values of ROW and N. ROW can be any of the relevant rows between ROWA and ROWJ inclusively, while N is the position of the cell in this row.



The two adjacent rows are defined as LAST-ROW and NEXT-ROW. These can, of course, be obtained directly from the property list of ROW using the identifiers BEFORE and AFTER, but it makes the coding easier if specific names are chosen. In the diagram, standard LISP notation has been chosen to show the numerical position of the sites. These values can now be used directly in the new function NEIGHBOURS.

```
(DEFUN NEIGHBOURS (ROW N)
 (SETF LAST-ROW (GET ROW 'BEFORE))
 (SETF NEXT-ROW (GET ROW 'AFTER))
 (SETF COUNT 0)
 (DOTIMES (X 3 T)
          (SETF P (+ X (1- N)))
          (IF (NTH P (EVAL LAST-ROW))
           (ADD-COUNT) T)
          (IF (NTH P (EVAL ROW))
           (ADD-COUNT) T)
          (IF (NTH P (EVAL NEXT-ROW))
           (ADD-COUNT) T))
(IF (NTH N (EVAL ROW)) (1- COUNT) COUNT))
```

First, the variables ROW and N need to be placed upon the parameter list. The neighbouring rows are defined by the two symbolic expressions (SETF LAST-ROW (GET ROW 'BEFORE)) and the analogous (SETF NEXT-ROW (GET ROW 'AFTER)). The number of neighbours of the site at ROW, N, is initialised by the further expression (SETF COUNT 0).

The DOTIMES loop which follows can then check simultaneously the corresponding sites in all three of the rows, LAST-ROW, ROW and NEXT-ROW. This is done in a vertical slice across the dish defined by the value of P. P is related to the central site, and the value of the loop variable, by (SETF P (+ (X (1- N))) in a way which ensures that all three of the columns shown on the diagram are examined in turn.

Three conditional expressions are involved in the investigation of the nine sites. For example, on the first pass through the DOTIMES loop the first of the conditional expression in the loop, (IF (NTH P (EVAL LAST-ROW)) (ADD-COUNT) T), checks the top left hand site shown on the diagram. On the second pass, it examines the middle site of the top row. EVAL has to be used once more to evaluate LAST-ROW to the list of 20 elements. NTH then selects the appropriate element according to the value of P. If this site contains a cell, the value of (NTH P (EVAL LAST-ROW)) will be T and the function ADD-COUNT is called. This increments the value of COUNT by 1.

```
(DEFUN ADD-COUNT NIL
 (SETF COUNT (1+ COUNT)))
```

After the DOTIMES loop in NEIGHBOURS has been completed, every filled site has contributed to COUNT. This will give a maximum theoretical value of 9, because the central site itself will have been included. Clearly, this is a potential error in the function, as the site is not one of its own neighbours. If the central site does contain a cell, the overall value must therefore be reduced by one. This is achieved by the final expression in NEIGHBOURS, which returns the adjusted value of COUNT if the central site is filled, or simply COUNT if it is not.

Simulating a Biological Process - CYCLE

This is the function which embodies the essential simulation of BACTERIA.

```
(DEFUN CYCLE NIL
 (SETF ROW 'ROWA)
 (DO ((X 1 (1+ X)))
     ((NULL ROW))
     (DOTIMES (Y 20 T)
              (NEIGHBOURS ROW Y)
              (COND ((OR (< COUNT 2) (> COUNT 3))
                     (ALTER (GET ROW 'COPY)
                      (1+ Y) NIL))
                    ((= COUNT 3) (ALTER
                     (GET ROW 'COPY) (1+ Y) T))
                    (T NIL)))
     (SETF ROW (GET ROW 'AFTER)))
 (DOTIMES (X 10 T)
          (SET (NTH X CELLS) (EVAL (GET
           (NTH X CELLS) 'COPY)))))
```

First, ROW is given the initial value of ROWA to begin the examination of the dish at the top line of the display. The exit condition (NULL ROW) in the DO loop which follows is triggered when the final row has been checked, in a similar way to the operation of the earlier function DISPLAY-CELLS.

The DOTIMES loop looks at each element of ROW in turn. The use of (NEIGHBOURS ROW Y)) at each site provides a value of COUNT which is used in the COND expression. The first clause of this tests for extinction of a cell. The second permits the creation of a fresh cell. Thus, if the value of COUNT is less than 2, or more than 3, the first clause of the COND places a NIL in the corresponding location of ROW's copy in the duplicate set of cells, COPIES. This is achieved by the use of the function ALTER in (ALTER (GET ROW 'COPY) (1+ Y) NIL). The appropriate row in COPIES is found by using GET and the position in the row is determined by incrementing Y by 1. This is necessary because the loop counts the elements from an initial zero, while the function ALTER begins with the first element labelled as one. This made the use of ALTER in the function SEED more convenient for the person operating the program.

In a similar fashion, the second clause of the COND uses ALTER to place a value of T in COPIES for any site where a new cell needs to be created. Thus, if (= COUNT 3) has the value T, the expression (ALTER (GET ROW 'COPY) (1+ Y) T) adjusts ROW's copy accordingly.

After the DOTIMES loop, (SETF ROW (GET ROW 'AFTER)) selects the next row in CELLS to be examined. Finally, a further DOTIMES loop, following the DO loop, transfers all the new values recorded in COPIES back into CELLS for the next time-step. This cannot be done until this stage. Any earlier updating of COPIES could easily cause spurious results, because different regions of the dish would belong simultaneously to different time-steps. EVAL needs to be included in this last loop to transfer the actual 20 element lists into CELLS and not just their identifiers.

Assembling the Complete Program

With the addition of CYCLE, the whole simulation can now be assembled as the final function BACTERIA.

```
(DEFUN BACTERIA NIL
 (SPACE)
 (FORMAT T "-------- BACTERIA --------")
 (TERPRI)
 (DISH)
 (DISPLAY-CELLS)
 (SPACE)
 (FORMAT T "Type initial number of cells.")
 (TERPRI)
 (SETF N (READ))
```

```
(DOTIMES (X N T)
         (TERPRI)
         (FORMAT T "Cell No. ")
         (PRIN1 (1+ X))
         (SEED))
(TERPRI)
(DISPLAY-CELLS)
(DOTIMES (X 10 T)
         (SET (NTH  X COPIES) (EVAL
          (NTH X CELLS))))
(SPACE)
(FORMAT T "Do you wish to see a cycle?")
(DO ((X 1 (1+ X)))
    ((NULL (Y-OR-N-P)))
    (CYCLE)
    (SPACE)
    (FORMAT T "Cycle No. ")
    (PRIN1 X)
    (TERPRI)
    (DISPLAY-CELLS)
    (SPACE)
    (FORMAT T "Do you wish to see a cycle?"))
    (FORMAT T "Simulation ended."))
```

After the simulation has been begun by typing (BACTERIA), the function referred to in Chapter 6, SPACE, is employed to help create a clear display.

```
(DEFUN SPACE NIL
 (TERPRI)
 (TERPRI))
```

TERPRI is then used before DISH initialises CELLS and COPIES. After this, DISPLAY-CELLS shows the ten empty rows on the monitor. The user is now prompted to add cells to the dish with the message shown by (FORMAT T "Type initial number of cells."). The number entered is assigned to N, which in turn controls the use of SEED inside the DOTIMES loop. The overall result is that the second call to DISPLAY-CELLS shows the dish with its initial colony of cells.

Before the development of the cells in the dish can be simulated in the DO loop which follows, the contents of CELLS must first be transferred to the duplicate rows held in COPIES. This is done by (SET (NTH X COPIES) (EVAL (NTH X CELLS))) in the second DOTIMES loop

The user is given the option of selecting each cycle in turn, rather than the program automatically displaying the next stage after a set delay. This is necessary to allow careful analysis of the results obtained. It is achieved in BACTERIA by controlling exit

from the DO loop with the condition (NULL (Y-OR-N-P)). If the user's response to 'Do you wish to see another cycle?' is negative, then (Y-OR-N-P) will have the value NIL and thus (NULL (Y-OR-N-P)) will evaluate as T. The loop then ends.

Within the loop, SPACE, TERPRI and FORMAT T are all used to create an effective display. At each pass through the loop, CYCLE determines the new cell population which is subsequently shown by DISPLAY-CELLS. Finally, outside the DO loop the message 'Simulation ended' brings the program to its close.

Typical Execution of BACTERIA

A typical execution of BACTERIA is now shown. Two distinct cell populations are placed into the dish. One is the example already investigated above. It provides a useful test here, showing that the simulation does behave as anticipated. It is at this stage of writing a simulation that thorough debugging may well become critical and very time-consuming! Note that when the chosen cells are typed into the program there must be no gap between ROW and the relevant letter, although the use of EXPLODE in DISPLAY-CELLS does include this gap in the print-out of the dish.

A second cell distribution has also been seeded into the culture medium. This group of five cells, with its distinctive 'crawl' across the dish, is an example of the interesting discoveries that can be made with a simulation program. Other arrangements of cells can also be tested far more rapidly than they could have been without the program. In fact, the time spent in creating a suite of functions, like those which comprise BACTERIA, can always be regained with interest as the simulation is used to analyse the development through each successive time-step of various initial conditions selected by the user.

```
* (BACTERIA)

-------- BACTERIA --------

ROW A .....................
ROW B .....................
ROW C .....................
ROW D .....................
ROW E .....................
ROW F .....................
ROW G .....................
ROW H .....................
ROW I .....................
ROW J .....................

Type initial number of cells.
8
Cell No.1
```

```
Type row, then position.
ROWG 3
Cell No.2
Type row, then position.
ROWG 4
Cell No.3
Type row, then position.
ROWG 5
Cell No.4
Type row, then position.
ROWG 13
Cell No.5
Type row, then position.
ROWG 14
Cell No.6
Type row, then position.
ROWG 15
Cell No.7
Type row, then position.
ROWH 5
Cell No.8
Type row, then position.
ROWI 4

ROW A ..................
ROW B ..................
ROW C ..................
ROW D ..................
ROW E ..................
ROW F ..................
ROW G ..OOO.......OOO.....
ROW H ....O..............
ROW I ...O...............
ROW J ..................

Do you wish to see a cycle? (Y or N) Yes

Cycle No.1

ROW A ..................
ROW B ..................
ROW C ..................
ROW D ..................
ROW E ..................
ROW F ...O.........O......
ROW G ...OO........O......
ROW H ..O.O........O......
ROW I ..................
ROW J ..................
```

```
Do you wish to see a cycle? (Y or N) Yes
Cycle No.2

ROW A ....................
ROW B ....................
ROW C ....................
ROW D ....................
ROW E ....................
ROW F ...OO...............
ROW G ..O.O.......OOO.....
ROW H ....O...............
ROW I ....................
ROW J ....................

Do you wish to see a cycle? (Y or N) Yes

Cycle No.3

ROW A ....................
ROW B ....................
ROW C ....................
ROW D ....................
ROW E ....................
ROW F ...OO........O......
ROW G ....OO.......O......
ROW H ...O.........O......
ROW I ....................
ROW J ....................

Do you wish to see a cycle? (Y or N) Yes

Cycle No.4

ROW A ....................
ROW B ....................
ROW C ....................
ROW D ....................
ROW E ....................
ROW F ...OOO..............
ROW G .....O......OOO.....
ROW H ....O...............
ROW I ....................
ROW J ....................

Do you wish to see a cycle? (Y or N) No
NIL
```

Summary

1. LISP can be used in computer simulations.

2. System dynamics is an approach to writing a simulation which analyses possible causal loops within a system distinct from its environment.

3. EXPLODE and IMPLODE are useful functions to add to the LISP environment.

4. MAPCAR takes a whole list as its argument and applies a given function to each element, returning another list.

5. A lambda expression can be used in conjunction with MAPCAR.

6. SET also performs assignments but evaluates both its arguments.

7. BUTLAST removes elements from the end of a list.

8. NTHCDR removes elements from the beginning of a list.

9. EVAL forces an extra stage of evaluation.

10. LAST returns the final element of a list as a single element list.

Exercise 7

1. Use MAPCAR together with a lambda expression to square the elements of a numerical list.

2. Use BUTLAST and NTHCDR to return the middle of a list without its first, and last, two elements.

8

Seeing the Light: Computer Vision and Pattern Recognition

The development of computer programs which are able to recognise patterns is a very important area of research. The problem is difficult, as witnessed by the fact that a considerable proportion of the human cortex is given over to the analysis of visual images, which, of course, is fundamentally a question of pattern recognition. In fact, it has been suggested that, really, seeing is one of the most 'intelligent' things that we can do.

Giving computers and robot machines even a limited ability to see would increase their efficiency enormously. Factory robots with sophisticated vision systems would be far more versatile and not require their environment to be so closely controlled. Office machinery capable of reading hand-written documents would save enormously on labour costs. Robot space probes sent to the other planets would be able to navigate intelligently and even decide for themselves which features of their locality were worth further examination. NASA scientists have already designed a prototype Martian robot vehicle which, when tested in an area of rough terrain on Earth, was able to steer itself safely from one place to another without colliding with large boulders deliberately left in its path.

Pattern Recognition in LISP

In this chapter, a function will be developed which is able to recognise geometric shapes. The actual analysis of signals from a video camera is, of course, well beyond the scope of this book but the principles outlined here are quite relevant to such a task, though on a smaller scale. The assumption will be made that initial analysis of an image has already reduced it to a set of pixels represented on a co-ordinate array. To reduce the time taken in the analysis, the array is also kept fairly small. Nevertheless, the program is quite powerful and able to remember the simple 'images' given to it. It is able to cope with changes in the orientation of the image as well as differences in size.

Before this program is begun, however, a simple pattern recognition function is written which will be later incorporated into the main LISP program. This function, `ALIKE`, will match one list against another and produce a numerical value. The bigger the number generated, the closer the two lists are matched.

```
(DEFUN ALIKE (A B)
 (SETF M 0)
 (SETF N (IF (> (LENGTH A) (LENGTH B))
  (LENGTH A) (LENGTH B)))
 (DOTIMES (X N T)
          (IF (EQ (NTH X A) (NTH X B))
           (SETF M (1+ M)) T))
 (/ M N))
```

ALIKE is deliberately contrasted with a very similar function, AKIN. The reason for this will be seen shortly.

```
(DEFUN AKIN (A B)
 (SETF M 0)
 (SETF N (IF (> (LENGTH A) (LENGTH B))
  (LENGTH A) (LENGTH B)))
 (DOTIMES (X N T)
          (IF (EQ (NTH X A) (NTH X B))
           (SETF M (1+ M)) T))
 M)
```

The operation of ALIKE is quite easy to understand. The two lists being compared are A and B. A variable, M, is used to give the score indicating how closely the lists match. This is first bound to zero by (SETF M 0). A second variable, N, is then made equal to the length of the longer of the two lists by the expression (SETF N (IF (> (LENGTH A) (LENGTH B)) (LENGTH A) (LENGTH B))). This value for N is then used to control the DOTIMES loop which follows.

The loop matches the elements in the two lists in pairs. M is incremented when corresponding elements are the same. It is at this stage that the two versions of the function differ. AKIN simply returns the value of M, whereas ALIKE divides it by the value of N. Although this might seem to be of little importance, it can be seen to make a great difference in how well the two functions perform when tested in their ability to match two lists. This can be observed in these examples.

```
* (ALIKE '(A B C) '(A B C))
1.0

* (AKIN '(A B C) '(A B C))
3

* (ALIKE '(A A) '(A A A A))
0.5

*(AKIN '(A A) '(A A A A))
2
```

AKIN tends to find closer similarity between lists than they deserve. This is particularly true in the case of the second example. AKIN would find as much matching between, say, (A A) and (A A) as it would between (A A) and (A A A A A A). This is clearly undesirable. ALIKE, simply by the addition of the division, is capable of distinguishing in far finer detail and for this reason is the version of the function employed in the main program below.

Simulating Computer Vision - RETINA

A simple simulation of computer vision can now be constructed. This program, RETINA, will begin at a fairly late stage in the interpretation of a visual image and makes the assumption that a scene has already been reduced to a rectangular grid of pixels. The further assumption is invoked that only two levels of brightness are involved. The pixels may be either switched on or switched off. The picture therefore suffers from unrealistically harsh contrast. The shades of grey which would normally contribute to an interpretation of an image are excluded from the first. Whilst being unrealistic for real applications, this simplification is essential in a relatively unsophisticated program operated by a microcomputer.

Processing a genuine image requires a great deal of computational power. Even a small increase in the size of the image, or the amount of detail required, will rapidly multiply the total number of pixels which have to be investigated. This in turn has a considerable effect upon the number of individual calculations which must be made in relating one pixel to those around it. Analysing images is a particularly good example of the combinatorial explosion in computing. This is the general problem of a small increase in the complexity of a problem leading to a far greater increase in the difficulty of its analysis, rather like the apocryphal story of the Buddhist monk who modestly suggested as his reward one grain of rice, on the first square of the chess board he had recently invented for a grateful king, two on the second, four on the third... In order to avoid unduly long operating time, RETINA only attempts to analyse images on a 15x15 grid. Although this is small enough to permit rapid processing of the simple images presented to the program, the area is still large enough to investigate relatively complicated shapes.

RETINA is able to identify the vertices of a two-dimensional shape by locating those shaded pixels which are not simply co-linear with the pixels surrounding them. The number of vertices discovered, and their mutual distances apart, form pattern which is used to generate a unique identification code. This can be compared with other codes which the program has already encountered and, if the shape has been named at an earlier stage by the user, RETINA will suggest this name again. Alternatively, if the code cannot be matched, the program will request a new name to associate with it. In this way, RETINA becomes an example of a simple 'machine learning' system.

The efficiency of the learning which takes place depends upon several factors, which will be considered in a later section. However, it can be seen immediately that not least amongst them will be the ability of the pattern matching algorithm to cope with the

examples with which it is presented. RETINA will therefore test the function ALIKE, developed above, in a taxing, near real-world situation. It will be seen that ALIKE, and the other algorithms employed in RETINA, are able to cope with shapes which are moved around the grid, enlarged and even rotated.

The functions which need to be written for RETINA are the following. PIXELS defines the 15x15 grid upon which the whole program depends. This is displayed on the screen by SHOW-PIXELS. The functions SKETCH and LIGHT-UP allow the user to design a shape upon the grid. After this, the vertices of the shape are identified by a new predicate function, VERTEXP. The user is informed about the vertices with LOCATE and RECORD. An identification code for the current shape being analysed is calculated by CODE-SHAPE and compared with the existing codes and names by RECOGNISE. Finally, the entire set of functions are controlled by the supervisory function itself, RETINA.

In addition, the functions developed for RETINA will require several of those already defined earlier in this text. And so before the whole suite of programs can be operated, SPACE, PAUSE, ABC, SQUARE, INTEGER, STRIP-ZERO, ORDER, GREATEST, REMOVE, IMPLODE and ALIKE will all need to be included in the current LISP environment.

Initialising the Retina - PIXELS

This function is needed to establish the 15 lists which will store all the basic information about the grid and the shape drawn by the user for analysis. Fortunately, this part of RETINA can borrow extensively from the functions developed for the simulation, BACTERIA, in the previous chapter. As in the function employed then, DISH, the lists recording the data are all elements themselves of an overall list. In this function, this master list is GRID. As before, it is introduced to simplify the coding of the relationships between the individual sublists.

```
(DEFUN PIXELS NIL
 (SETF GRID '(RA RB RC RD RE RF RG RH RI RJ RK RL
  RM RN RO))
 (DOTIMES (X 14 T)
          (PUTPROP (NTH X GRID) (NTH (1+ X) GRID)
           'AFTER)
          (PUTPROP (NTH (1+ X) GRID) (NTH X GRID)
           'BEFORE))
 (SETF RA NIL)
 (DOTIMES (X 15 T)
          (SETF RA (CONS NIL RA)))
 (MAPCAR #'(LAMBDA (R) (SET R RA)) GRID))
```

First, SETF is used to establish GRID as the 15 lists RA, RB, RC ... RO. A DOTIMES loop then relates these lists to one another via their property lists so that, for example, (GET RB 'BEFORE) evaluates to the preceding list RA and (GET RB 'AFTER) to the following list RC. This is directly analogous to the use of property lists earlier in DISH.

The second DOTIMES loop then sets up the first list RA as a series of 15 NILS. Each of these corresponds to a pixel in the top row of the grid in its initial state. A value of NIL represents a pixel which has not yet been switched on. Following this, (MAPCAR #'(LAMBDA (R) (SET R RA)) GRID) makes identical lists for RB, RC ... RO.

Revealing the Image - SHOW-PIXELS

A shape for analysis can now be placed into GRID. In order for the appropriate points of the shape to be identified, the set of 225 pixels is placed on the screen by SHOW-PIXELS. Each row and column of the array of points is labelled with a number and a letter respectively.

```
15 . . . . . . . . . . . . . . .
14 . . . . . . . . . . . . . . .
13 . . . . . . . . . . . . . . .
12 . . . . . . . . . . . . . . .
11 . . . . . . . . . . . . . . .
10 . . . . . . . . . . . . . . .
9  . . . . . . . . . . . . . . .
8  . . . . . . . . . . . . . . .
7  . . . . . . . . . . . . . . .
6  . . . . . . . . . . . . . . .
5  . . . . . . . . . . . . . . .
4  . . . . . . . . . . . . . . .
3  . . . . . . . . . . . . . . .
2  . . . . . . . . . . . . . . .
1  . . . . . . . . . . . . . . .

   A B C D E F G H I J K L M N O
```

The way in which the pixels are to be selected is deliberately like the game of 'battleships'. This is intended to make the choice of points particularly easy for the user.

```
(DEFUN SHOW-PIXELS NIL
 (TERPRI)
 (SETF R 'RA)
 (DO ((X 0 (1+ X)))
     ((NULL R))
     (TERPRI)
     (PRIN1 (- 15 X))
     (IF (> 6 X) (FORMAT T " ") (FORMAT T "  "))
     (DOTIMES (Y 15 T)
              (IF (NTH Y (EVAL R)) (FORMAT T "X")
               (FORMAT T "."))
              (FORMAT T " "))
     (SETF R (GET R 'AFTER)))
 (SPACE)
 (FORMAT T "   A B C D E F G H I J K L M N O"))
```

(SETF R 'RA) first binds the identifier R to the first of the lists in GRID, RA. RA is quoted so that R is the symbol RA, rather than the list (NIL NIL NIL...). This ensures that GET can be used later in the function to obtain the next list in GRID, RB, but it also means that EVAL will be needed in the function for an extra stage of evaluation.

The DO loop begins with a value of zero for the variable X. The expression (PRIN1 (- 15 X)) then conveniently labels each row of pixels on the screen display with the correct number. An immediate problem is that the numbers greater than 9 are, naturally enough, two digits long. This would shift the top six lines of the display one character space to the right in an annoying fashion unless a correcting factor were introduced. This is the conditional expression which directly follows the PRIN1, (IF (> 6 X) (FORMAT T " ") (FORMAT T " ")). For the first six rows of the display, (> 6 X) evaluates as T and so the first action of the conditional takes place. One extra character space will then follow the two digit row number. With the remaining rows of the grid, (> 6 X) is NIL and so the second action results. Now two spaces are printed after the single digit row number, automatically adjusting the display.

The DOTIMES loop nested within the DO loop evaluates each pixel of the row in turn with (NTH Y (EVAL R)). If the value is T, an X is printed on the display. This represents a chosen pixel. If the value is NIL, a full-stop is added, to register the position of the unselected pixel. In either case, the use of (FORMAT T " ") after the conditional expression places an extra empty space after every pixel in the row. This makes the display square rather than rectangular.

The last expression of the DO loop is (SETF R (GET R 'AFTER)). This obtains the next list in GRID to be investigated for its pixel content. When the condition (NULL R) has resulted in the loop ending, the earlier function (SPACE) places a gap on the display before FORMAT T is used to print out the letters which identify the columns.

Selecting Individual Pixels - LIGHT-UP

The function LIGHT-UP allows the choice of individual pixels which will be part of the selected shape on the grid. First, attention must be given to the fact that the labelling of the rows and columns of the grid in SHOW-PIXELS yields a clear display, but does not correspond very closely with the way in which the pixels are to be coded into LISP. LIGHT-UP needs to be able to convert from the user's co-ordinates into the relevant LISP representation. This requires that the letter typed in selecting a row must be converted into a number which can be employed in the expressions involving NTH.

Similarly, the number entered when identifying a row needs to be interpreted as one of the lists contained in GRID. The necessary function is coded as on the following page.

```
(DEFUN LIGHT-UP NIL
 (TERPRI)
 (FORMAT T "Type the column letter...")
 (SETF C (- (FIRST (COERCE (STRING (READ))
'LIST)) 65))
 (TERPRI)
 (FORMAT T "Type the row number......")
 (SETF R (NTH (- 15 (READ)) GRID)))
```

The three dots following the first prompt, 'Type the column letter', are included to make the display easier to read. They lead to the letter entered by the user falling into the same column of the screen display as all other entries made. The character chosen is accepted as a string by the expression (STRING (READ)) and then coerced to a numerical list containing the ASCII code of the letter by the symbolic expression (COERCE (STRING (READ) 'LIST)). FIRST is used to remove the brackets and 65 subtracted to reduce the ASCII value to the number which is assigned to the symbol C and which represents the desired position in one of the lists of GRID. For example, if A is typed, (STRING (READ)) will be (A). (COERCE (STRING (READ)) 'LIST) will have the value (65). The expression (FIRST (COERCE (STRING (READ)) 'LIST)) evaluates as 65 and finally (- (FIRST (COERCE (STRING (READ)) 'LIST)) 65) is 0. This value is the first element of any of the rows in GRID.

Next, LIGHT-UP requests the row number. Six dots are now included after the prompt for a neatly formatted screen display. The number typed by the user can be converted into the required row very directly. If the user types 1, the intended row is R0 which is the 14th element of GRID, as the function NTH always begins counting at zero. Similarly, if 2 is entered, the row required is RN and this is the 13th list embedded in GRID. The symbolic expression required is therefore (NTH (- 15 (READ)) GRID) and this value is assigned to the identifier R.

LIGHT-UP only allows one pixel on the grid to be selected by the user and does not actually adjust the value of this location in GRID. Therefore it is only part of an overall function, SKETCH, which uses LIGHT-UP in the construction of a shape.

Drawing a Shape on the Grid - SKETCH

This is the routine which enables a shape to be traced on the grid of 225 pixels prior to its analysis by the subsequent functions within RETINA.

The routine is shown on the next page:

```
(DEFUN SKETCH NIL
 (SPACE)
 (FORMAT T "Type number of points....")
 (SETF N (READ))
 (DOTIMES (X N T)
          (TERPRI)
          (FORMAT T "Point No.")
          (PRIN1 (1+ X))
          (LIGHT-UP)
          (SET R (APPEND (BUTLAST (EVAL R)
           (- 15 C)) (CONS T (NTHCDR (1+ C)
             (EVAL R))))))
 (SHOW-PIXELS))
```

After SPACE has placed a gap on the display, FORMAT T requests the number of points needed for the shape that is going to be drawn. Inevitably, this means that some thought needs to be given beforehand to the arrangement of the points on the grid, but specifying the total number of points in advance does simplify the structure of the function.

The number of points chosen is assigned to N, a value then controlling the DOTIMES loop which follows. Then FORMAT T informs the user, with (PRIN1 (1+ X)), which pixel is currently being chosen by the call to LIGHT-UP. Here, appropriate values for C and R are obtained. After this the symbolic expression (SET R (APPEND (BUTLAST (EVAL R) (- 15 C)) (CONS T (NTHCDR (1+ C) (EVAL R))))) alters the value of the pixel identified by LIGHT-UP to T to show that it is now part of the shape. The expression is closely similar to the corresponding code in the earlier function ALTER. There are some differences caused by the fact that the parameter N in ALTER assumed that counting began at 1 rather than zero. This means that (1+ C) must be substituted in the NTHCDR and BUTLAST expressions used in ALTER. Additionally, there are now 15 rather than 20 elements in the list. When these two alterations are made, the expression used in ALTER becomes the one employed here in SKETCH.

At the end of the DOTIMES loop, the completed design is displayed by a further call to SHOW-PIXELS, which is now able to use the values of T inserted into the sub-lists of GRID to print a capital letter X at all the positions defining the shape.

With the four functions, PIXELS, SHOW-PIXELS, LIGHT-UP and SKETCH, the program RETINA can store any simple polygon which can be defined on a 15x15 array of dots. The analysis of the shape, and its identification, will now depend upon the further functions, beginning with VERTEXP.

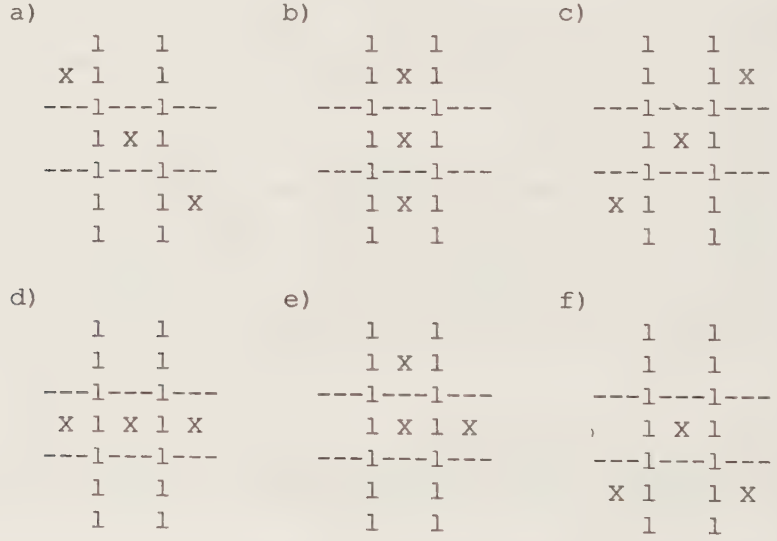
Identifying a Vertex - VERTEXP

A vital algorithm in RETINA is the one which is capable of detecting vertices of shapes on the grid. This is coded into LISP as the function VERTEXP. It is named like

a typical predicate function because it returns a value of T or NIL, depending upon whether the point of the grid being considered is a vertex or not.

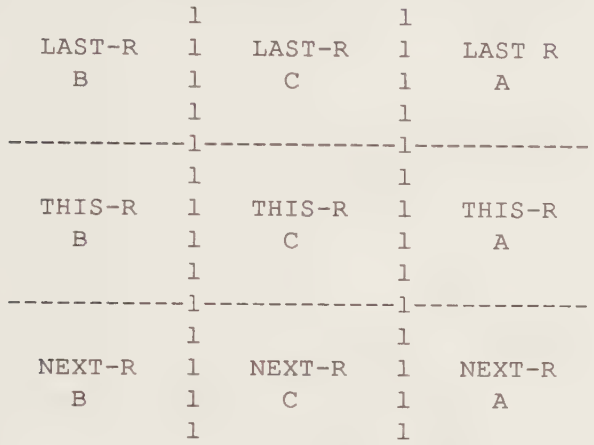
As always in programming, the finest points of detail must be stated in a completely explicit way before any attempt is made to convert them into program code. Therefore before a function can be written to identify vertices, the nature of 'vertex' in the current context must be carefully considered. Everybody has a firm, intuitive concept of what is meant by a vertex of a two-dimensional shape which has been drawn in the conventional way on a sheet of paper. But here a continuous two-dimensional surface is being used, as well as continuous one-dimensional lines. The intersection of these defines the vertices of the shape.

A different situation emerges with the limited nature of the grid utilised in RETINA. Now there are just 225 pixels, arranged in a square array of 15 rows and columns. The 'surface' is not continuous and neither are the 'lines' drawn upon it. In this context, the definition of a vertex must be re-established. It can be done by considering a single pixel and the eight pixels around it. In these diagrams, a lighted pixel is represented by a capital X and an unlighted pixel by an empty space.



In each diagram, the central, lighted pixel has been allocated two lighted neighbours. There are many possible arrangements of the two pixels about the central position. However, only the four combinations shown in diagrams a) - d) appear to match an intuitive concept of continuity. In these diagrams, the central pixel lies on a straight line, but in diagrams e) and f) it is clearly at, or near, a vertex. In e) the diagram could show the corner of a square or a rectangle and in f) the diagram might represent a vertex of a triangle.

In the function VERTEXP, the definition of continuity and discontinuity implied by the diagrams is coded into LISP to identify which lighted pixels are vertices. Any pixel which forms part of the shape and which does not correspond with one of the first four diagrams is assumed to be a vertex. First, the pixels which are neighbours of the central pixel are identified according to this more detailed diagram. For uniformity, the central position is described as row THIS-R and column C.



The function can now be written to use these identifiers.

```
(DEFUN VERTEXP (R C)
 (SETF LAST-R (EVAL (GET R 'BEFORE)))
 (SETF THIS-R (EVAL R))
 (SETF NEXT-R (EVAL (GET R 'AFTER)))
 (SETF B (1- C))
 (SETF A (1+ C))
 (NULL (OR (AND (NTH B LAST-R) (NTH A NEXT-R))
           (AND (NTH C LAST-R) (NTH C NEXT-R))
           (AND (NTH A LAST-R) (NTH B NEXT-R))
           (AND (NTH A THIS-R) (NTH B THIS-R)))))
```

VERTEXP has the variables R and C on its parameter list, for the row and column of the pixel being investigated. The rows LAST-R, THIS-R and NEXT-R are then all defined, using the property list of R where necessary. Similarly, the columns B and A are identified using C.

The long s-expression which follows then returns the value of T or NIL required. Each embedded AND will evaluate as T when one of the conditions shown in the four diagrams applies. If the central point is a vertex, none of the ANDs returns T and so the OR evaluates as NIL. The overall function then returns T. However, if the central point is not a vertex then one of the ANDs, and hence the OR expression, will have the value

T. VERTEXP returns NIL in this case. The function thus acts as a predicate which detects vertices of the shape as required.

Locating all the Vertices - LOCATE

The function VERTEXP can now be used in the function LOCATE to discover all the vertices of the shape. An additional function needed is RECORD.

```
(DEFUN LOCATE NIL
 (TERPRI)
 (SETF VERT-ROW (MAKE-ARRAY 10))
 (SETF VERT-COL (MAKE-ARRAY 10))
 (SETF V -1)
 (SETF R 'RA)
 (PAUSE)
 (DO ((X 0 (1+ X)))
     ((NULL R))
     (DOTIMES (Y 15 T)
              (IF (AND (NTH Y (EVAL R))
               (VERTEXP R Y)) (RECORD) T))
     (SETF R (GET R 'AFTER)))
 (TERPRI))
```

Two arrays, VERT-ROW and VERT-COL, are set up to store the row and the column of every vertex found during the search. An assumption is made that no more than ten vertices will be located. Naturally, this number could be altered easily if it were considered necessary. The current number of vertices identified is recorded by the variable V. This is used to update VERT-ROW and VERT-COL in the function RECORD. It is also incremented by one each time RECORD is called. As the first position in the arrays, and hence the first value of V required, is 0, it is necessary for V to be initialised as -1 in LOCATE by the expression (SETF V -1). Then after V has been incremented in the first call to RECORD it will have the correct array value.

The variable R is assigned to the first row of the grid by the expression (SETF R 'RA). Then, before the DO loop, the function PAUSE, which was written in Chapter Four, is included. This is quite important and is needed to allow the existing display to be studied before LOCATE begins searching for vertices.

The DO loop considers each row of the grid in turn and the nested DOTIMES loop examines every pixel in each row. At the end of the loop, (SETF R (GET R 'AFTER)) selects the next row. When all 15 have been checked, this expression returns a value of NIL for R and so the loop exits at (NULL R). The conditional within the loops, (IF (AND (NTH Y (EVAL R)) (VERTEXP R Y)) (RECORD) T), will call the function RECORD only when pixels are found which

belong to the shape as well as returning T for VERTEXP. If the expression (NTH Y (EVAL R)) were not included, LOCATE would decide that every empty pixel of the grid was a vertex.

Accountability in RETINA - RECORD

The function RECORD is coded like this.

```
(DEFUN RECORD NIL
 (TERPRI)
 (SETF V (1+ V))
 (FORMAT T "Vertex No.")
 (PRIN1 (1+ V))
 (FORMAT T " located at:")
 (TERPRI)
 (FORMAT T "Row No.")
 (PRIN1 (- 15 X))
 (FORMAT T " and Column ")
 (PRIN1 (AREF LETTER Y))
 (SETF (AREF VERT-ROW V) X)
 (SETF (AREF VERT-COL V) Y))
```

The purpose of RECORD is two-fold. It has to inform the user where vertices have been found, as well as storing their positions in the two arrays. These are employed in the coding algorithms later in RETINA. Presenting the information in the way this function does is significant. It is desirable that 'intelligent' software should be able to inform the user precisely what it is doing whenever possible. The currently fashionable phrase used to describe this aspect of software engineering is 'accountability' and it features widely in Artificial Intelligence circles. Basically, it provides a fail-safe. In drawing a shape with SKETCH, a stray point may have been included on another part of the grid. The user's mind may well automatically filter this out as being trivially important and, if this happens at a subconscious level, the extra pixel might not even be perceived by the user. However, it will inevitably be discovered by the function LOCATE. If RECORD then did not notify the user that it had been found, it would go on to be included in the coding routines and provide a false shape code and identity which would be carefully stored away for future, and erroneous, use.

The variable V is incremented by (SETF V (1+ V)) to permit fresh locations to be addressed in both of the arrays, VERT-ROW and VERT-COL. FORMAT T is then used to inform the user which vertex has been discovered. Here, (PRIN1 (1+ V)) temporarily increments V again because the user will not number the first vertex of the shape as the zero. The row of the vertex is found quite simply as (- 15 X), as can be checked from the example given of the grid's display on the monitor. The column of the vertex is converted to its letter value by the use of the array values in LETTER.

This is the reason why the earlier function ABC must be included in the current LISP environment.

Finally, RECORD fixes the location of the vertex on the grid by using the two arrays. The Vth location of VERT-ROW is assigned the X value of the DO loop in LOCATE. Similarly, the Y value from the DOTIMES loop is placed in VERT-COL. These two values can then be used in the next function, CODE-SHAPE.

Analysing the Vertex Pattern - CODE-SHAPE

CODE-SHAPE and its companion, RECOGNISE, are probably the most important functions within RETINA, providing the 'intelligence' of the program. CODE-SHAPE turns the locations of the vertices of the shape on the grid into a code unique for that shape, while RECOGNISE compares this code with the ones it already knows. It can then either name the shape or request what name should be associated with an unfamiliar code.

The algorithm utilised in CODE-SHAPE makes the assumption that the mutual distances of the vertices apart will automatically form a distinctive pattern for different shapes. This will be independent of the orientation of the shape. Additionally, if the distances are all expressed as ratios based upon the smallest distance, this pattern will be independent of the size of the shape. This algorithm is quite simple and yet powerful. In the example given below of an actual execution of RETINA, two shapes are identified as both being triangles. This correct identification of the second shape is made, even though it is more than twice the size of the first and rotated through 180 degrees.

The algorithm in CODE-SHAPE consists of the following stages. Suppose a square has been placed on the grid by the user. After the function LOCATE has operated on the information held in GRID, the four vertices of the shape will have ben detected and their positions stored in the arrays VERT-ROW and VERT-COL. The only part of the shape now under investigation will be these four points, perhaps these distances apart on the grid.

```
. . . . . .
. . . . . .
. . X . . . . X . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . X . . . . X . .
. . . . . .
. . . . . .
```

These are now examined in the new function, CODE-SHAPE. Two nested loops compare all the points with one another, calculating their distances from one another using the standard Pythagorean relationship. These are left as their squared values, because a code is produced quite satisfactorily without the need to find the square roots. With the square shown, a list of 16 numbers will be generated at this stage, although there will be only three distinct values. These are 0, where the distance of a vertex from itself is calculated, 49, corresponding to the side of the square and 98, representing a diagonal. These numbers must be simplified further before the code is complete. Zero values are simply ignored. This reduces the code to 12 numbers. All the numbers are divided by the smallest to give a set of multiples of the least value. By doing this, the code becomes capable of recognising shapes which vary in size. The problem of orientation of shapes is solved by placing all the numbers in order. After these modifications have taken place, the code for a square will be the list (1 1 1 1 1 1 1 1 2 2 2 2). Finally, these are converted to letters to give the code (A A A A A A A A B B B B). The reason for this is so that the function IMPLODE can operate on the list to present the user with the slightly briefer code "AAAAAAAABBBB". This compact version of the code will help in explaining what stage RETINA has reached in the analysis of the shape and is a further example of accountability in a program. The list itself is still used in the function RECOGNISE, because the matching function written earlier, ALIKE, requires lists as its arguments.

All these separate steps in producing the code are now interpreted as LISP expressions in the function CODE-SHAPE.

```
(DEFUN CODE-SHAPE NIL
 (TERPRI)
 (SETF SHAPE NIL)
 (SETF SMALL 20)
 (DOTIMES (X (1+ V) T)
          (DOTIMES (Y (1+ V) T)
                   (SETF D (+ (SQUARE (-
                    (AREF VERT-ROW X)
                     (AREF VERT-ROW Y)))
                      (SQUARE (- (AREF VERT-COL
                       X)
                        (AREF VERT-COL Y)))))
                   (IF (AND (< D SMALL) (> D 0))
                    (SETF SMALL D) T)
                   (IF (ZEROP D) T (SETF SHAPE
                    (CONS D SHAPE)))))
 (SETF SHAPE (MAPCAR #'(LAMBDA (A) (AREF LETTER
  (- (INTEGER (/ A SMALL)) 1))) (ORDER SHAPE)))
 (FORMAT T "Shape code is ")
 (PRIN1 (IMPLODE SHAPE)))
```

First, the variable SHAPE is assigned the value NIL. SHAPE will become the code for the shape, passed as a list to the next function, RECOGNISE. Another variable SMALL is introduced. This is needed in the part of the function which reduces the numbers produced to multiples of the smallest. It is given the initial value of 20, as this will be larger than the typical distances being calculated between vertices. SMALL will therefore be altered as soon as vertices are examined in the loops. It will always be adjusted to the smallest inter-vertex distance encountered. It can then be used in reducing all the numbers added to SHAPE to multiples of the least.

The two DOTIMES loops calculate the squared distance, D, between every vertex pair using the function SQUARE, written in Chapter Two, and the Pythagorean equation applied to the values held in the two arrays VERT-ROW and VERT-COL. The complete expression is

```
(SETF D (+ (SQUARE (- (AREF VERT-ROW X) (AREF VERT-ROWY)))
(SQUARE (- (AREF VERT-COL X) (AREF VERT-COL Y))))).
```

After this, the variable SMALL is readjusted to the value of D if this is the smaller number. The zero values of D which have been obtained when the distance of a vertex from itself has been measured are ignored by the extra clause in the condition, (> D 0), to generate the overall assigning expression for the variable SMALL, (IF (AND (< D SMALL) (> D 0)) (SETF SMALL D) T). Non-zero values of D are added to the shape code, SHAPE, by the further expression (IF (ZEROP D) T (SETF SHAPE (CONS D SHAPE))). Here, the second action of the conditional will only take place for the non-zero values of D which need to be added to the code.

At this point, the nested DOTIMES loops are completed. The code which has been produced consists of non-zero numbers, but they are neither ordered, nor reduced to multiples of the smallest. The code also needs to be converted to letters. All three of these operations are performed within a single symbolic expression that follows. This consists of a lambda expression applied to (ORDER SHAPE). ORDER is the function that was developed in Chapter Three and has the effect of placing all the numbers in ascending order. The lambda expression therefore just needs to obtain the ratios and convert these to letters. The ratios of each number to the smallest is found by the expression (INTEGER (/ A SMALL)). A is the variable used in the lambda expression. INTEGER is required because the value produced will be used to identify a location in the array LETTER. The smallest value which will be produced at this point is 1. However, as the array locations begin at 0, this would give B as the first number in any code. It is more plausible for codes to begin with A, so the value is decreased by 1. Finally, the array location is found by AREF. These additional requirements produce the lambda expression (LAMBDA (A) (AREF LETTER (- (INTEGER (/ A SMALL))))). The whole expression is applied to (ORDER SHAPE) by MAPCAR and this produces the final version of the code SHAPE.

It is this list which is used to identify shapes. However, the compacted string corresponding to the code is presented to the user by the use of IMPLODE, together with FORMAT T and PRIN1 to display the code.

Comparing with Known Shapes - RECOGNISE

This function will accept a given code and check it against all those already stored. From these it will select the one which matches the new code most closely and return the name which has been linked with this previous code by the user. All known codes are stored in the list GEOMETRY. Here are located the identifying sublists and corresponding names of the respective shapes, as a series of pairs of elements. In each pair, the code precedes the relevant name. At the beginning of a program run GEOMETRY will be simply be the list (NIL NIL), but as shapes are analysed, and their names typed in, it grows in complexity. If the first shape analysed is a square, and if, after it has been allocated a code, the user enters the name 'SQUARE', GEOMETRY becomes the list ((A A A A A A A A B B B B) SQUARE NIL NIL).

Because the codes and names alternate in GEOMETRY, the loop employed in RECOGNISE has to jump in twos through the list. This is done with a DOTIMES loop where the number of loop cycles is equal to half the length of GEOMETRY and where the loop variable is doubled before being used to identify elements of the list. The function is coded as follows.

```
(DEFUN RECOGNISE NIL
 (TERPRI)
 (SETF BEST-MATCH 0)
 (SETF LAST-MATCH 0)
 (SETF S (STRIP-ZERO (/ (LENGTH GEOMETRY) 2)))
 (DOTIMES (X S T)
          (SETF NEW-MATCH (ALIKE SHAPE
           (NTH (* 2 X) GEOMETRY)))
          (IF (> NEW-MATCH LAST-MATCH)
           ((LAMBDA NIL
            (SETF LAST-MATCH NEW-MATCH)
             (SETF BEST-MATCH (* 2 X)))) T))
 BEST-MATCH)
```

The variables BEST-MATCH and LAST-MATCH are first introduced. BEST-MATCH records the numerical position within GEOMETRY which the most closely matching code occupies. LAST-MATCH records the current best value, returned by the matching function ALIKE, for the code as it is compared with all those stored in GEOMETRY. The variable S is assigned a value equal to half the length of GEOMETRY. This is then used to control the DOTIMES loop which searches through the list. STRIP-ZERO is needed to produce a value for S which has no decimal part, as required by the DOTIMES variable.

In the loop, the variable NEW-MATCH is set equal to the matching value yielded by ALIKE as it compares in turn each code in GEOMETRY with the target code SHAPE. The conditional expression which follows then records the best matching value by constantly comparing NEW-MATCH with LAST-MATCH and always reassigning LAST-MATCH the higher value where necessary. This is achieved in the lambda expression, which also assigns BEST-MATCH the position in GEOMETRY of the code being checked. This results in BEST-MATCH always recording the position of the current most closely matching code. At the end of the function, before the ultimate bracket, BEST-MATCH reveals the location of the closest match found.

Assembling RETINA's Component Functions

With this final function RECOGNISE, the overall program can now be constructed.

```
(DEFUN RETINA NIL
 (SETF GEOMETRY '(NIL NIL))
 (SPACE)
 (FORMAT T "------------ RETINA ------------")
 (SPACE)
 (FORMAT T "Do you want to draw a shape?")
 (TERPRI)
 (DO ((X 1 (1+ X)))
     ((NULL (Y-OR-N-P)))
     (PIXELS)
     (SHOW-PIXELS)
     (SKETCH)
     (LOCATE)
     (CODE-SHAPE)
     (RECOGNISE)
     (FORMAT T "Is it a ")
     (PRIN1 (NTH (1+ BEST-MATCH) GEOMETRY))
     (TERPRI)
     (IF (Y-OR-N-P) T ((LAMBDA NIL (TERPRI)
      (FORMAT T "Please type name.") (TERPRI)
       (SETF GEOMETRY (CONS SHAPE (CONS (READ)
        GEOMETRY))))))
     (SPACE)
     (FORMAT T "Do you want to draw a shape?")
     (TERPRI))
 (SPACE)
 (FORMAT T "Program ended."))
```

First, RETINA's basic knowledge of shapes, the list GEOMETRY, is assigned its initial value of (NIL NIL). Then SPACE and FORMAT T are used to place the title on the screen and give a prompt to the user, 'Do you want to draw a shape?' This

prompt also needs to be placed at the end of the DO loop controlling the program. The predicate function (Y-OR-N-P) allows the user to exit the loop whenever desired.

Here are recapitulated the tasks which the functions perform in the loop. PIXELS sets up the lists needed to represent the grid of pixels, which is then displayed by SHOW-PIXELS. SKETCH lets the shape be drawn. The vertices are found by LOCATE and used to create the code in CODE-SHAPE. RECOGNISE finally chooses the nearest code in GEOMETRY. At this stage, RETINA needs to check its answer against the name the user would suggest for the shape. The name stored in GEOMETRY for the code located by BEST-MATCH is (NTH (1+ BEST-MATCH) GEOMETRY). This is presented to the user with the prompt 'Is it a'. (Y-OR-N-P) is used again. If 'N' is typed, a lambda expression then requests what the name of the shape should be. This is then added to the list GEOMETRY, together with the new code, by the remaining part of the lambda expression (SETF GEOMETRY (CONS SHAPE (CONS (READ) GEOMETRY))).
The operation of RETINA can now be demonstrated. This example was extracted from a program run in which RETINA had already been taught the names 'line' and 'square'.

```
* (RETINA)

Do you want to draw a shape?
 (Y or N) Yes

------------ RETINA ------------

15 . . . . . .
14 . . . . . .
13 . . . . . .
12 . . . . . .
11 . . . . . .
10 . . . . . .
9  . . . . . .
8  . . . . . .
7  . . . . . .
6  . . . . . .
5  . . . . . .
4  . . . . . .
3  . . . . . .
2  . . . . . .
1  . . . . . .

   A B C D E F G H I J K L M N O

Type number of points....9
Point No.1
Type the column letter...G
Type the row number......7
Point No.2
```

```
Type the column letter...G
Type the row number......8
Point No.3
Type the column letter...G
Type the row number......9
Point No.4
Type the column letter...G
Type the row number......10
Point No.5
Type the column letter...H
Type the row number......9
Point No.6
Type the column letter...I
Type the row number......8
Point No.7
Type the column letter...J
Type the row number......7
      Point No.8
      Type the column letter...I
      Type the row number......7
Point No.9
Type the column letter...H
Type the row number......7

      ------------ RETINA ------------

15 . . . . . .
14 . . . . . .
13 . . . . . .
12 . . . . . .
11 . . . . . .
10 . . . . . X . . . . . .
9  . . . . . X X . . . . . .
8  . . . . . X . X . . . . . .
7  . . . . . X X X X . . . . .
6  . . . . . .
5  . . . . . .
4  . . . . . .
3  . . . . . .
2  . . . . . .
1  . . . . . .

   A B C D E F G H I J K L M N O

    Vertex No.1 located at:
    Row No.10 and Column G
    Vertex No.2 located at:
    Row No.7 and Column G
    Vertex No.3 located at:
    Row No.7 and Column J
```

```
    Shape code is "AAAABB"
    Is it a SQUARE
     (Y or N) No
    Please type name.
    TRIANGLE

Do you want to draw a shape?
 (Y or N) Yes
------------ RETINA ------------

15 . . . . . .
14 . . . . . .
13 . . . . . .
12 . . . . . .
11 . . . . . .
10 . . . . . .
9  . . . . . .
8  . . . . . .
7  . . . . . .
6  . . . . . .
5  . . . . . .
4  . . . . . .
3  . . . . . .
2  . . . . . .
1  . . . . . .

   A B C D E F G H I J K L M N O

Type number of points....15
Point No.1
Type the column letter...I
Type the row number......13
Point No.2
Type the column letter...J
Type the row number......13
Point No.3
Type the column letter...K
Type the row number......13
Point No.4
Type the column letter...L
Type the row number......13
Point No.5
Type the column letter...M
Type the row number......13
Point No.6
Type the column letter...N
Type the row number......13
Point No.7
Type the column letter...N
Type the row number......12
```

```
Point No.8
Type the column letter...N
Type the row number......11
Point No.9
Type the column letter...N
Type the row number......10
Point No.10
Type the column letter...N
Type the row number......9
Point No.11
Type the column letter...N
Type the row number......8
Point No.12
Type the column letter...M
Type the row number......9
Point No.13
Type the column letter...L
Type the row number......10
Point No.14
Type the column letter...K
Type the row number......11
Point No.15
Type the column letter...J
Type the row number......12

---------- RETINA ----------

15 . . . . . .
14 . . . . . .
13 . . . . . X X X X X X .
12 . . . . . X . . . X .
11 . . . . . X . . X .
10 . . . . . X . X .
9  . . . . . X X .
8  . . . . . X .
7  . . . . . .
6  . . . . . .
5  . . . . . .
4  . . . . . .
3  . . . . . .
2  . . . . . .
1  . . . . . .

   A B C D E F G H I J K L M N O

Vertex No.1 located at:
Row No.13 and Column I
Vertex No.2 located at:
Row No.13 and Column N
Vertex No.3 located at:
```

```
Row No.8 and Column N

Shape code is "AAAABB"
Is it a TRIANGLE
 (Y or N) Yes

Do you want to draw a shape?
 (Y or N) No

Program ended.
```

Summary

1. Pattern recognition is important in A.I. and easily coded into LISP.

2. Realistic computer vision systems require a great deal of computational power but simple models can be developed on a personal computer running LISP.

3. Recognising edges and vertices is an important early stage in processing a visual image.

4. Pattern matching is also important in teaching a program to learn from experience.

9

Knowledge to Order: Expert Systems

Expert systems represent a highly significant application of the principles of Artificial Intelligence in the ordinary world. The emergence of A.I. from the university research department into real life has largely been a consequence of the applicability of expert systems to practical problems. At the same time a great stimulus for further research, and the necessary funding from industry, resulted from the commercial application of expert systems.

Expert systems have the immense advantage of providing ready access to specialist knowledge of the sort which usually would be only available if the genuine human expert were present. They allow non-specialists to process information and make decisions which they would not normally be able to. They allow unlimited duplication of the real expert and also the rather chilling possibility of a person's expertise continuing after the death of the individual. Really they represent yet another facet of the curiously human ability to extend a single person's ability across society as a whole, partly the driving principle behind the evolution of human culture.

There are disadvantages to expert systems as well. They take time to develop and can be expensive. As yet, they still require a more powerful computer than the average micro, although they are just beginning to appear in the personal computer market. Expert systems are also clearly better adapted to certain limited ranges of human information. They are not, at present, a universal tool that can be applied to any problem. Additionally, in the last analysis, the ultimate decisions have to be made by the actual people who use the expert system as a tool which offers advice and reasons for its opinion. Handing total control over to the machine is inadvisable. Proper decisions require a human personality with emotions and not the self-referencing, automatic encyclopaedia to which an expert system approximates. Unfortunately, there does seems to be a trend, within the world of international finance as well as military planning, to regard the expert system as infallible and almost superhuman.

Representing Knowledge in an Expert System

The efficient operation of an expert system will depend very much upon the way in which its information is stored. Indeed, the origin of this new category of computer software followed the discovery at the Massachusetts Institute of Technology, by the team led by Joel Moses, that the inclusion of practical, heuristic information into A.I.

software led to a breakthrough in problem-solving. The way in which this information is made available to the system is extremely important. Currently, there are four main methods of knowledge representation employed. These are frames, scripts, semantic networks and production rules. They can be used separately or in combination with one another.

Frames

A frame is a table of information on a particular subject. Individual entries are called slots. Four types of slot may be incorporated into a frame. One type simply states a particular piece of information appropriate to the subject. Another type, a default slot, will contain an inevitable piece of information. For example, in any frame referring to a cat the slot giving the number of legs will default to four. A procedural attachment slot defines a routine or procedure needed to determine further information for the frame. Finally, a reference slot links the current frame with another which contains relevant further information. Reference slots allow a hierarchy of frames to be constructed, thus building up a broad knowledge-base. Here are two linked frames which could form part of a knowledge-base about a cat living in a factory.

```
PUSSY CAT FRAME

Species: Cat

Name: Toddy

Number of legs: 4 (default)

Hobby: Sleeping

Activity: (Reference Factory Cat)

FACTORY CAT FRAME

Work place: Shop floor

Home base: Caretaker's hut

Employment: Chasing mice

Salary: Cat food

Additional benefits: Milk
```

Scripts

A script is very much like a frame, in that it stores detailed and fairly specific information. Unlike a frame, however, it describes a process rather than specific

subjects. Variations in a script are 'tracks'. 'Roles' are the principal characters involved and 'props' are objects. 'Scenes' relate the actual process in order. 'Entry conditions' trigger this part of the overall script. 'Results' show the final situation and may match the entry conditions which will trigger another track of the script. Keeping to the same example, part of a typical script could be as follows.

```
CHASE SCRIPT

Track: Mouse eliminated

Roles: Cat
       Mouse

Props: Floor
       Mouse hole
       Television
       Machinery
       Saucer of milk

Entry conditions: Mouse leaves hole.
                  Cat is awake.

Scene 1: Detection
Mouse leaves hole.
Cat observes mouse.
Mouse does not see cat.
Mouse runs across floor.

Scene 2: Chasing
Cat jumps off television.
Cat stalks mouse.
Mouse sees cat.
Mouse runs.

Scene 3: Reward
Cat catches mouse.
Mouse squeaks.
Cat has crisis of conscience.
Mouse leaves factory in gratitude.

Results
Factory's product is uncontaminated.
Caretaker gives cat milk.
```

Semantic networks

A semantic network is an easily comprehended way of representing information. It is simply a network of nodes containing related items linked by arcs representing their

relationship. Figure 9.1 is an example of a semantic network holding information, again about part of the same situation used above.

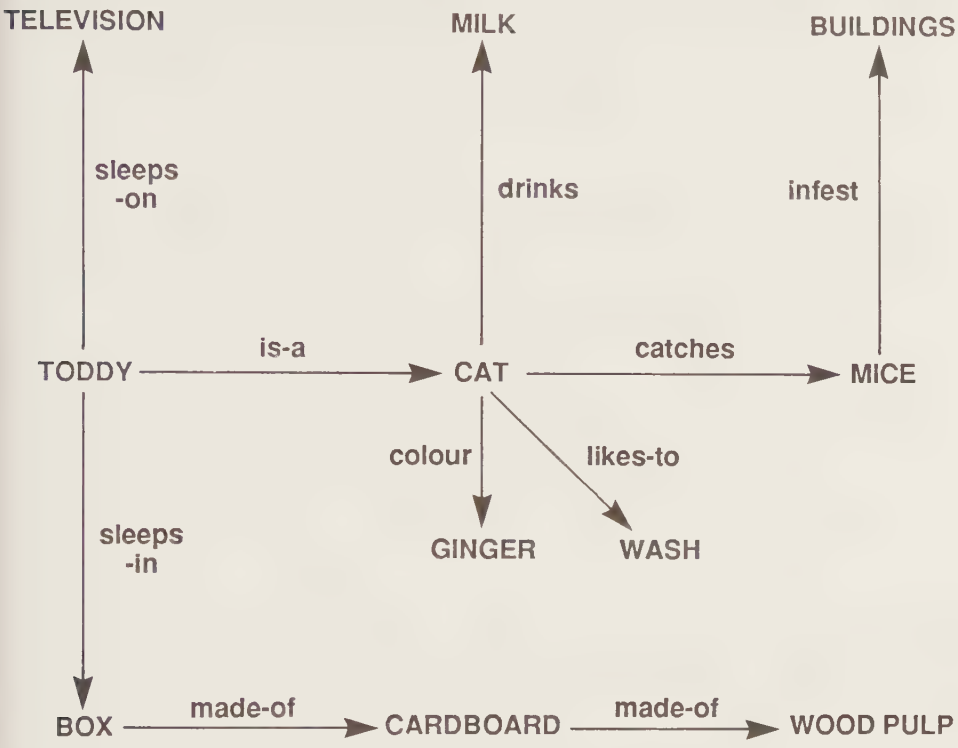


Figure 9.1

The basic principles of a semantic network have already been explored in the program ARISTOTLE in Chapter Five. It is fairly obvious that the network can be incorporated in a very useful way into an expert system and allow sensible decisions to be made.

Production rules

The method most often used for storing information in an expert system is to include a large set of `IF-THEN` clauses, known as production rules. These allow sequences of decisions to be made and logical consequences to be inferred. Production rules can be especially easy to understand as they reflect a human way of reasoning. Here, for example, is the rule-base which will be needed for an expert system which can identify characters on children's television. Naturally, these are characters currently appearing on British television, including elderly repeats on Channel 4, and so they should be familiar to most readers, especially those who either have children of their own or who tend to leave the television on regardless. The list of personalities is necessarily

limited, but should be comprehensive enough to indicate how a rule-base operates within an expert system. Altogether 12 characters from children's television could be identified using this rule base. They are Doctor Who, Batman, Geoffrey, Lassie, Black Beauty, Popeye, Tom, Sylvester, Gordon, Spotty Dog, Sam and Captain Scarlet.

```
1. IF   The character is alive
   AND  The character has two legs
   THEN The character is human

2. IF   The character is human
   AND  The character keeps changing identity
   THEN The character is Doctor Who

3. IF   The character is human
   AND  The character uses shark repellent
   THEN The character is Batman

4. IF   The character is human
   AND  The character knows Zippy and George
   THEN The character is Geoffrey

5. IF   The character is alive
   AND  The character has four legs
   THEN The character is an animal

6. IF   The character is an animal
   AND  The character barks
   THEN The character is Lassie

7. IF   The character is an animal
   AND  The character neighs
   THEN The character is Black Beauty

8. IF   The character is not alive
   AND  The character is two dimensional
   THEN The character is a cartoon

9. IF   The character is a cartoon
   AND  The character eats spinach
   THEN The character is Popeye

10. IF   The character is a cartoon
    AND  The character is a cat
    AND  The character likes mice
    THEN The character is Tom

11. IF   The character is a cartoon
    AND  The character is a cat
    AND  The character likes canaries
    THEN The character is Sylvester
```

```
12. IF   The character is not alive
    AND  The character is three dimensional
    THEN The character is a puppet

13. IF   The character is a puppet
    AND  The character squeaks
    AND  The character is really a sock
    THEN The character is Gordon

14. IF   The character is a puppet
    AND  The character does a silly walk
    THEN The character is Spotty Dog

15. IF   The character is a puppet
    AND  The character is a member of
         the down-trodden masses
    AND  The character helps Daddy Woodentop
    THEN The character is Sam

16. IF   The character is a puppet
    AND  The character is indestructible
    THEN The character is Captain Scarlet
```

The Inference Engine

The part of the expert system which does the reasoning is known as the 'inference engine'. This draws upon both the stored knowledge and replies from the user of the system in order to reason its way through to an answer. In a production rule system, two types of inference can be made, forward chaining, and backward chaining. In forward chaining the inference engine begins with the information currently provided by the user and draws conclusions, according to the conditional rules that it knows already. During this process, it may request further details from the user. Eventually, it will arrive at logical consequences, which it then gives as its decision. In backward chaining, the system begins with the required answer and then searches through its production rules to seek out what prior conditions would be required. Again it eventually arrives at a set of ultimate clauses which are necessary for the final state and it seeks to match these against the details provided by the user. Of course, as should be familiar from the work done earlier with the program ARISTOTLE, the path of true conditionals which will be followed, the relevant arcs in the network, can become very complicated. Nevertheless, the algorithms employed by the inference engine will be able to cope with such complexity.

The Need for Machine Learning

A problem faced in building expert systems is found in entering all the necessary information required for its decision making. It is a long and very tedious process to obtain all of the knowledge required from a human expert. Mistakes can be made in transferring data from mind to program. Repeated adjustment will be required to the

expert system in order to check that the new rules are behaving as expected. This will inevitably be very time consuming.

In a more subtle way, many of the vital processes involved in the human expert's decision making may not actually be obvious to the person involved. It is a familiar paradox that the more competent we become at a skill the less we understand precisely what we are doing when we exercise it. There is the familiar nursery story of the centipede who could cope perfectly well until another arthropod politely asked it how it co-ordinated the movement of all its legs. Similarly, although a child learning how to ride a bicycle could probably explain quite well to somebody else what had to be done to avoid falling off, an adult would probably find it far more difficult to make a formal list of those particular skills which were involved. This is essentially one of the problems facing anybody who is trying to code a human expert's skills into computer software. Many of the necessary details will be so automatic for the expert that they will lie buried deep beneath the surface and be very hard to make explicit.

Instead, the ideal expert system would develop its own knowledge-base, initially asking broad questions and then narrowing its research down into fine details as it understood more about the domain in which it was operating. As it learnt enough to realise where its ignorance was most evident, and what extra points had to be clarified for it, it could ask the human expert for the help needed at that stage. Care is needed with the concept of 'understanding', as applied to a machine, and this will be need to be investigated further in a subsequent section.

Of course, expert systems of this sophistication still have to be developed, but will form a vital step forward into true machine intelligence. This is because, looking into the distant future, another important role for computer software capable of intelligent learning will be in controlling robots. At the moment the sort of mechanical man which has existed for over 50 years in fiction still seems only a distant probability. However, with the constant refinememt of new computer architectures, especially those where parallel processing takes place over a distributed system of interlinked smaller units, and with the continual research that is being carried out in Artificial Intelligence software, it does seem inevitable that robots will eventually become a reality. Such quasi-human machines will inevitably require an ability to learn independently of their creators. It will be quite impossible to program such devices in such a way that all possible situations in which they find themselves will be amenable to analysis and interaction on the basis of some earlier algorithm permanently recorded into their inherent computational ability. Instead, they will need to modify and adjust their behaviour and 'mental' reactions according to the type of experience they encounter. Just as the human infant has the ability to learn how to cope with the world, without the whole of its neural network being inflexibly committed at birth, so the robot of the future will require sophisticated software capable of learning from the machine's experience.

The Importance of Feedback

A very simple component of the learning process, and yet one which is easily overlooked, is the presence of feedback. Basically, unless somebody tells you when

you are going wrong you are unlikely ever to improve upon your performance. This is probably one of the reasons why self-tuition can be so difficult. It also figures significantly in the distinction between a good personal tutor and a poor one. If essays are read and marked carelessly, or not at all, the student will inevitably suffer. There is nothing oddly psychological about this. It will be seen in the LISP application which follows that no learning takes place when errors are not corrected.

Learning by Small Amounts

Another vital aspect of learning is that it cannot take place if the gulf between knowledge and ignorance is too large. A first school child, however alert with the sand tray, is going to pale at the inclusion of tensor calculus into the 'integrated day'. In the same way, it is a mistake for the programmer to expect an expert system to cope with data hopelessly, or perhaps even slightly, beyond its current horizon. Interestingly, this mistake was made in the early days of artificial intelligence. With the heady optimism of pioneers, the first A.I. researchers planned problem-solving software of remarkable prowess. The proposed diversity of these programs never materialised.

Learning and Understanding

Part of the current concept of 'learning' is understanding. It is generally assumed that a person does not really learn something new until he or she understands it thoroughly. Such understanding is implicitly regarded as an essential component of any learning process. This attitude has certainly been prevalent for at least the last two decades in junior education in Britain, where the previous faith in the efficacy of rote learning has been constantly ridiculed. At times this has, possibly, been taken to extremes. Thus, some teachers have been very reluctant to encourage their children to memorise multiplication tables until a very thorough and deep understanding of the fundamental principles of arithmetic has been acquired through constant manipulation of structural apparatus.

Meanwhile other people associated with the educative process, frequently the parents of the children involved, have suspected that fundamentally an incorrect premise has been assumed. They would claim that whereas it is quite pleasant for a young mind to have a clear grasp of basic set theory, the cash value of accurate mental arithmetic is perhaps greater in many practical situations.

Fortunately, when considering learning as a machine process the situation is much clearer. As no human mind is involved, the debatable psychological advantages of understanding can be ignored completely. All a piece of computer software designed to learn need achieve is a response of some kind when the appropriate input is provided. This response may be the return of new or processed information, or some other reaction. Thus, if the machine is asked, 'What is the capital of France?', it will reply 'Paris' without any need for additional information flashing through its circuits about

political and geographical dependence, sociological and economic theory, or the history of Western Europe since the collapse of the Roman Empire.

This is specifically a black box situation. Certain inputs are provided and other outputs, of the appropriate sort, obtained. What happens inside the black box is irrelevant, provided that it works. The glass box of contemporary educational theory need not be applied to machine learning at present, though it will probably become more important in the future when the robots and androids begin at last to be designed.

There is no solid reason for disquiet in a position like this. Although as humans we all feel much happier when we seem to understand the fundamental justification for our beliefs, a black box model of learning only reflects the opinion of David Hume, the 18th century Scottish philosopher. In his attack on the principle of induction, he observed that we never have any logical ground for even our most cherished beliefs. All we can observe is the continual succession of linked events. To use a familiar example, the constant observation of white swans does not prove that all swans are white. No 'understanding' of some deep, fundamental association has been established. And, of course, early settlers in Australia did indeed discover black swans. Hume's view of knowledge should provide comfort for any computer scientist who is attempting to place knowledge into an expert system. Perhaps what is achieved in the machine is not too unlike the processes occurring in the human mind.

Simulating Knowledge Acquisition

The points discussed above about learning and knowledge acquisition will now be incorporated practically into a set of LISP functions which do simulate on a small scale the way in which experience can be codified into a knowledge-base. It will be seen how important feedback is to this process. In particular, without the presence of initial errors which are then corrected, no learning will take place at all. The irrelevance for a computer of genuine understanding in the human sense will also be demonstrated. The final program, `ORACLE`, is able to make accurate decisions based upon its experience of previous situations, but cannot be said to understand why it is able to do this. The knowledge it gains is, in fact, coded in a very transparent, arithmetical way. Yet to somebody unfamiliar with the operation of the functions involved, `ORACLE` may appear remarkably intelligent. This is similar, perhaps, to the way in which Joseph Weizenbaum's famous ELIZA was able to trick the unknowing into supposing it really cared about their psychological problems.

A Simple Learning System

There need be nothing too mysterious about setting up a system which will exhibit most of the required characteristics that can be reasonably expected of 'learning'. Provided that the area of application of the program is made sufficiently limited, it can learn to make correct decisions on the basis of the evidence given to it and to offer its own judgment. The most important feature of the program is that no initial expert

knowledge has to be fed into it. In a purely automatic fashion, the program learns to adjust its internal representation of a situation according to whether the decisions being made by the program are correct or not. Once the program's knowledge has reached a final state of equilibrium, all the decisions made will be correct.

The main disadvantage of this otherwise ideal program is that its area of expertise is extremely limited. In the form in which it will be developed below, it is only capable of distinguishing between two possible states or decisions. It could, for example, give advice of the form 'Buy 100 shares in Octopus Conglomerates PLC' or, alternatively, 'Don't buy 100 shares in Octopus Conglomerates PLC'. It would not be able to share its opinion with you about the advisability of purchasing 200 shares, considering another company's floatation, or buying a new stereo instead.

Nevertheless, the program does illustrate a method in which learning can be simulated in a mechanical way and remove some of the mystique surrounding the general subject of expert systems.

The operation of the learning system can be explained best if an extremely simple example is considered. In this, the two possible answers that the program can give are `LONDON` or `EDINBURGH`. Additionally, the program will ask only one question in order to select the city it should give as its answer. This question will be 'Is it located in Scotland?'. The question could, of course, be anything which will distinguish between the two places. For example, the program could ask 'Is it fairly close to the Thames?' or 'Was it originally a volcano?' The important point to note is that the program does not contain at first any explicit information linking the answers given to questions like these to the actual identity of the city. After an initial dialogue of the form:

```
Is it located in Scotland?
Yes
```

The program might still make the apparently idiotic reply:

```
Is it LONDON?
```

Until it learns that this sequence receives a terse, negative retort from the user, there is no reason why such an incorrect reply should not be given. The program is, after all, learning. It must be allowed to make mistakes, just as a person does when unfamiliar with the task at hand.

The learning system in this initial example is, therefore, limited to just two possible answers and one question. Later the number of questions will be increased, although the system will still only distinguish between two possible decisions. The third simplification is the way in which its knowledge is represented. In this example, it is restricted to a single variable which may take only three values `1`, `0` or `-1`. This will be called the 'rule' variable.

It will be seen subsequently that the two non-zero values correspond with the two possible answers, LONDON or EDINBURGH. The zero value is equivalent to 'Don't know', when the program is incapable of making the correct decision. Initially, the variable has zero as its value. When it has learnt how to select its answer, the system will have assigned one of the other two possible values to the rule variable. The way in which this value is given to the rule variable has, of course, still to be explained. It is closely related, as would be expected, to the way in which the system decides between its two answers.

The system makes its decision in a purely arithmetical way. After asking the user its question, it assigns a value to the reply typed. A positive answer is given the value 1 and a negative answer receives the value 0. The product of this value and the current value of the rule variable is then found. For an affirmative answer the calculation will be performed in this way.

```
Is it located in Scotland?
 (Y or N) Yes

REPLY   VALUE   RULE   RULE X VALUE

 YES      1      0          0

A negative reply will lead to:

Is it located in Scotland?
 (Y or N) No

REPLY   VALUE   RULE   RULE X VALUE

 NO       0      0          0
```

Note that at this stage, when the system is still ignorant and the rule variable has the value 0, the final result of the calculation is the same for either reply from the user.

The program now makes its decision upon the value of the product RULE X VALUE. If this is zero or positive, the system selects the first answer LONDON. If the product is negative, EDINBURGH is chosen. This decision making is, of course, purely arbitrary and would not advance the system's 'knowledge' about London and Edinburgh by the slightest amount. However, a crucial stage in the learning algorithm occurs at this point. The user is asked whether the answer returned by the system is correct or not. If it is correct, no adjustment is made to the rule variable. If the answer is incorrect, then the rule variable is altered. At this point feedback is occurring in the learning process. It is an explicit example of learning taking place from negative instances rather than positive, as a corollary of the dictum 'You only learn from defeat'. The adjustment obeys a simple relationship with the product RULE X VALUE. If this is positive, it is subtracted from the rule. If it is negative, it is added to the rule. In each case, a new value for the rule variable results from an incorrect decision.
Here is this learning algorithm applied to the current example. The assumption is made that the user is thinking about London when answering the program's question.

```
Is it located in Scotland?
 (Y or N) No

REPLY VALUE  RULE  RULE X VALUE
 NO      0     0          0

Positive value implies LONDON
Is this correct?
 (Y or N) Yes
```

For simplicity, the program has regarded zero as positive quantity.

Although the system has given the correct answer, this is deceptive and happened by chance. No learning takes place as a result of this exchange and the value of the rule variable remains in its zero state of ignorance. The fact that the system lacks basic knowledge is seen from this alternative example, in which the user is thinking about Edinburgh and not London.

```
Is it located in Scotland?
 (Y or N) Yes

REPLY  VALUE  RULE  RULE X VALUE
 YES     1     0          0

Positive value implies LONDON
Is this correct?
 (Y or N) No
```

This is a critical stage because the negative response from the user triggers the alteration of the rule variable.

```
Rule needs modification
VALUE RULE RULE - VALUE
  1     0          -1
```

The program has given an incorrect answer and as a consequence the value of the rule variable is adjusted. This new value will now be used in the next decision made. It will be seen that with the adjusted rule the program can make the right decision in both cases. In this first example, the city the user is referring to is London.

```
Is it located in Scotland?
 (Y or N) No

REPLY VALUE RULE  RULE X VALUE
 NO      0    -1          0

Positive value implies LONDON
Is this correct?
 (Y or N) Yes
```

Here, where the user was thinking about London, the program was able to provide the appropriate answer. Similarly, the rule works if the user chooses Edinburgh.

```
Is it located in Scotland?
 (Y or N) Yes

REPLY   VALUE   RULE   RULE X VALUE

 YES      1      -1          -1

Negative value implies EDINBURGH
Is this correct?
 (Y or N) Yes
```

The learning algorithm works in exactly the same way if a series of questions is provided. The rule now becomes an array of numbers, the value of each of which will be initially zero but which may become 1 or -1, or perhaps remain as zero, when the system has finished its learning process. The decision will now be made on the basis of the total of individual products of the separate replies and the corresponding variables in the rule array. Below is shown how the system will function when there are three questions and three numbers in the array. Initially, the user is referring to Edinburgh again.

```
Is it fairly close to the Thames?
 (Y or N) No
Was it originally a volcano?
 (Y or N) Yes
Is it a tourist attraction?
 (Y or N) Yes
REPLY   VALUE   RULE   RULE X VALUE
 NO       0      0          0

 YES      1      0          0

 YES      1      0          0

Total is 0 + 0 + 0 = 0

Positive value implies LONDON
Is this correct?
 (Y or N) No

Rule needs modification

VALUE   RULE   RULE - VALUE
  0      0          0

  1      0         -1

  1      0         -1
```

The rule is adjusted further in this second example, in which the correct answer would be London.

```
Is it fairly close to the Thames?
 (Y or N) Yes
Was it originally a volcano?
 (Y or N) No
Is it a tourist attraction?
 (Y or N) Yes

REPLY   VALUE   RULE   RULE X VALUE

 YES      1       0          0

 NO       0      -1          0

 YES      1      -1         -1

Total is 0 + 0 + -1 = -1

Negative value implies EDINBURGH

Is this correct?
 (Y or N) No

Rule needs modification

VALUE   RULE   RULE + VALUE

  1       0          1

  0      -1         -1

  1      -1          0
```

This new rule array proves to be the final, correct version which will inevitably yield correct decisions. As said originally, a positive array value corresponds to that question requiring the first possible decision as its subject. A negative array value is associated with the second decision as the subject of the question. A value of zero means that the question provides no fresh information distinguishing between the two.

Here the user is referring to Edinburgh.

```
Is it fairly close to the Thames?
 (Y or N) No
Was it originally a volcano?
 (Y or N) Yes
Is it a tourist attraction?
 (Y or N) Yes
```

```
REPLY   VALUE   RULE   RULE X VALUE

 NO       0       1          0

 YES      1      -1         -1

 YES      1       0          0

Total is 0 + -1 + 0 = -1

Negative value implies EDINBURGH

Is this correct?
 (Y or N) Yes
```

And in the next example the correct answer is London.

```
Is it fairly close to the Thames?
 (Y or N) Yes
Was it originally a volcano?
 (Y or N) No
Is it a tourist attraction?
 (Y or N) Yes

REPLY   VALUE   RULE   RULE X VALUE

 YES      1      1          1

 NO       0     -1          0

 YES      1      0          0

Total is 1 + 0 + 0 = 1

Positive value implies LONDON

Is this correct?
 (Y or N) Yes
```

It now remains to be shown how this learning algorithm can be coded into a series of LISP functions, `QUESTIONS`, `ASK`, `TABLE`, `OPINION`, `MODIFY` and the overall function, `ORACLE`.

Before the learning system can be developed as these LISP functions it is necessary to introduce another Golden Common LISP primitive, `READ-LINE`.

Common LISP Primitive - READ-LINE

When it is required to accept a keyboard entry from the user during the evaluation of a LISP function, `READ` can be employed to register the current word being typed. There is a disadvantage to this, however. If you are trying to enter a sentence with several

words, only the first will actually be assigned by READ to the variable you have chosen. Suppose, for example, you have typed in an expression like:

```
(SETF WORDS (READ))
```

and then try to enter:

```
Is it located in Scotland?
```

You will immediately discover that only the first word will be bound to WORDS.

```
* WORDS
Is
```

One solution to this would be to place hyphens between each word of the sentence:

```
Is-it-located-in-Scotland?
```

A neater solution, though, is to use the LISP function READ-LINE rather than READ. This does not automatically accept keyboard entry when it detects a character space. Instead, it waits until the return key has been pressed to enter an entire line. You would first type in the expression:

```
(SETF A (READ-LINE))
```

After this, everything that is typed before the return key is pressed will be assigned to A, as a string between quotes. READ-LINE will now be used in the first LISP function needed for the learning system, QUESTIONS.

Selecting the Criteria - QUESTIONS

This function is used to type in the two possible answers which the system learns to distinguish, together with the questions which it will ask the user in order to develop its rule array. The function also sets up the different LISP arrays which are needed during the operation of the overall system. It is coded as follows.

```
(DEFUN QUESTIONS NIL
 (TERPRI)
 (FORMAT T "Type first answer:")
 (TERPRI)
 (SETF ANSWER-A (READ))
 (TERPRI)
 (FORMAT T "Type second answer:")
 (TERPRI)
 (SETF ANSWER-B (READ))
 (TERPRI)
 (FORMAT T "Type number of questions:")
```

```
(TERPRI)
(SETF N (READ))
(TERPRI)
(SETF QUERY (MAKE-ARRAY N))
(SETF VALUE (MAKE-ARRAY N))
(SETF RULE (MAKE-ARRAY N))
(SETF PRODUCT (MAKE-ARRAY N))
(DOTIMES (X N T)
         (FORMAT T "Type question No:")
         (PRIN1 (1+ X))
         (TERPRI)
         (SETF (AREF QUERY X) (READ-LINE))))
```

First, TERPRI and FORMAT T are used to place the prompt 'Type first answer' on the display and to bind the user's choice to the variable ANSWER-A in the expression (SETF ANSWER-A (READ)). Only a single word answer is assumed here, although READ-LINE could be employed in the expression if a longer answer were required. After this a similar routine requests the second answer from the user and assigns this to ANSWER-B.

Next, the function needs to know the number of questions which will be asked. (FORMAT T "Type number of questions:") places the necessary prompt on the screen and the user's reply is bound to the variable N by (SETE N (READ)). This value for N is very important. It is used to set up four different arrays, QUERY, VALUE, RULE and PRODUCT, as well as controlling the DOTIMES loop which follows. QUERY is the array which stores the questions themselves. VALUE holds each set of replies from the user. RULE is the essential rule array which gradually learns how to make the correct decision. PRODUCT is used to record each set of products, RULE X ARRAY, during the program's operation and to display them on the screen.

The DOTIMES loop at the end of QUESTIONS asks the user to type in each question in turn. The number of the question is displayed by PRIN1, but is first incremented by one so that the numbering does not begin at zero. The questions are placed into the QUERY array by the use of READ-LINE, as explained above, in the final s- expression, (SETF (AREF QUERY X) (READ-LINE)).

QUESTIONS will operate like this, using the first example again:

```
* (QUESTIONS)
Type first answer:
LONDON
Type second answer:
EDINBURGH
Type number of questions:
```

```
3
Type question No:1
Is it fairly close to the Thames?
Type question No:2
Was it originally a volcano?
Type question No:3
Is it a tourist attraction?
```

Asking the Questions - ASK

The next function needed for the learning system is ASK. This will discover the user's response to each of the questions and so play a vital role in the program's decision making. It is coded as shown below:

```
(DEFUN ASK NIL
 (SETF TOT 0)
 (DOTIMES (X N T)
          (PRINT (AREF QUERY X))
          (TERPRI)
          (SETF (AREF VALUE X)
           (IF (Y-OR-N-P) 1 0))
          (SETF (AREF PRODUCT X) (* (AREF RULE X)
           (AREF VALUE X)))
          (SETF TOT (+ TOT (AREF PRODUCT X)))))
```

The variable, `TOT`, is established by `(SETF TOT 0)`. It is used to record the total of all the separate products of `RULE X VALUE` for each of the questions asked.

The `DOTIMES` loop then places each of the questions, held in the `QUERY` array, on the screen in turn with `(PRINT (AREF QUERY X))`. The user has to respond with a YES or NO answer. This is prompted by the use of `(Y-OR-N-P)` and so all that the user needs to do is press the Y or N key. The whole word does not have to be typed in.

The corresponding number to be stored in the `VALUE` array is `1` for an affirmative answer and `0` for a negative, as explained above. This is done quite conveniently by using the (Y-OR-N-P) predicate within a conditional expression, `(IF (Y-OR-N-P) 1 0)`. This has the value 1 when Y is pressed and 0 if N is selected. The array is then updated by `(SETF (AREF VALUE X) (IF (Y-OR-N-P) 1 0))`.

The function also calculates the correct value to be placed in the `PRODUCT` array. This will be the current value in the `RULE` array, `(AREF RULE X)`, multiplied by the value which has just been entered into the `VALUE` array, `(AREF VALUE X)`. This quantity is naturally found by evaluating `(* (AREF RULE X) (AREF VALUE X))`. The appropriate location in the `PRODUCT` array is then filled by `(SETF (AREF PRODUCT X) (* (AREF RULE X) (AREF VALUE X)))`.

Finally, ASK keeps the total of all the individual products by updating TOT with `(SETF TOT (+ TOT (AREF PRODUCT X)))`.

The final two closing brackets after this last expression correspond, of course, to the `DOTIMES` and the `DEFUN` and balance the overall function.

Illustrating the decision - TABLE

It is important that the program is able to explain how it makes its decision, according to the established A.I. principle of accountability. Its operation is far easier to understand if the mechanical reasoning involved, together with the continual updating of the rule array, is made quite explicit to the user. One of the functions which will help to explain the way in which decisions are made, and the rule array adjusted, is `TABLE`. This will display the user's reply, the numerical value linked with the reply, the rule array and the product of the rule and the value, in a clear, tabular form. The function produces a display like this typical extract:

```
REPLY   VALUE   RULE   RULE X VALUE

 YES      1      1          1

 NO       0      1          0

 YES      1      0          0
```

It is evident that the spacing involved in this display function is quite critical and the earlier function `BLANK` is used in order to arrange the entries neatly in each column. Here is the function coded into LISP:

```
(DEFUN TABLE NIL
 (SPACE)
 (FORMAT T "REPLY  VALUE  RULE  RULE X VALUE")
 (DOTIMES (X N T)
          (SPACE)
          (BLANK 1)
          (IF (ZEROP (AREF VALUE X))
           (FORMAT T "NO ")
            (FORMAT T "YES"))
          (BLANK 5)
          (PRIN1 (AREF VALUE X))
          (BLANK 5)
          (PRIN1 (AREF RULE X))
          (BLANK 9)
          (PRIN1 (AREF PRODUCT X))))
```

SPACE is the function which uses TERPRI twice to leave a gap of one free line on the display. After this, FORMAT T places the four headings on the next line. A DOTIMES loop follows to add a row of four entries to the table for each question asked.

(BLANK 1) places a space before the first entry in the reply column. The reply itself is then obtained from the VALUE array. A value of 0 will require NO to be printed in the reply column while a value of 1 will correspond to YES. Which of the two words will be added is decided by the conditional expression. If 0 is the value in the array, then (ZEROP (AREF VALUE X)) will evaluate as T and so (FORMAT T "NO ") displays the negative reply. If the value in the array is 1, (ZEROP (AREF VALUE X)) will be NIL and so the affirmative reply is added by (FORMAT T "YES"). Note the extra space after NO to make this string the same length as YES.

(BLANK 5) then ensures that the next entry is placed beneath the letter U in the heading, 'RULE'. This entry will simply be the numerical value of the user's reply, obtained as (AREF VALUE X) and displayed by (PRIN1 (AREF VALUE X)). The same gap is again provided by (BLANK 5), followed by the numerical entry in the rule array. This is (AREF RULE X).

A larger space is then required to put the final entry in the centre of the fourth column because the heading here is longer. This is achieved with (BLANK 9). The product of the rule and the value is taken from the PRODUCT array and printed by the final expression (PRIN1 (AREF PRODUCT X)).

Making the Decision - OPINION

This function uses the values in the final column of the table which has been displayed, the numbers in the PRODUCT array, to decide which of the two answers should be chosen in response to the user's replies. OPINION is defined as follows.

```
(DEFUN OPINION NIL
 (SPACE)
 (FORMAT T "Total is ")
 (DOTIMES (X (1- N) T)
          (PRIN1 (AREF PRODUCT X))
          (FORMAT T " + "))
 (PRIN1 (AREF PRODUCT (1- N)))
 (FORMAT T " = ")
 (PRIN1 TOT)
 (SPACE)
 (IF (< TOT 0) (FORMAT T "Negative")
  (FORMAT T "Positive"))
 (FORMAT T " value implies ")
 (IF (< TOT 0) (PRIN1 ANSWER-B)
  (PRIN1 ANSWER-A))
 (SPACE)
 (FORMAT T "Is this correct?"))
```

First, the total of all the entries in the PRODUCT array is displayed. This number has already been calculated as TOT in the function ASK. Now the addition is made explicit.

SPACE places a blank line on the display. The total of the individual products is introduced by (FORMAT T "Total is") and then a DOTIMES loop shows each number separately. The loop also shows the addition sign between each of the numbers, with the expression (FORMAT T " + "). Because the number of plus signs will evidently be one fewer than the number of array elements, the loop's repeating is determined by the value of (1- N). After the loop the final element in the array is shown by the expression (PRIN1 (AREF PRODUCT (1- N))). The equals sign is added by (FORMAT T " = ") and then the total with (PRIN1 TOT).

The decision is made according to whether the total is positive or negative. These alternatives are declared on the display by using the sign of the total to select either of the words 'Negative' or 'Positive' in the next s-expression in the function (IF (< TOT 0) (FORMAT T "Negative") (FORMAT T "Positive")).

This is followed by (FORMAT T " value implies ") and then the appropriate decision, ANSWER-A or ANSWER-B. These are chosen according to whether the total is less than zero, or either zero itself, or greater than zero, in the conditional expression (IF (< TOT 0) (PRIN1 ANSWER-B) (PRIN1 ANSWER-A)).

Finally, the program needs to know whether its decision is right. The relevant text is printed by (FORMAT T "Is this correct?") The user's response, however, is obtained in the main body of the program, the function ORACLE, and not in OPINION.

Adjusting the Rule - MODIFY

Learning will not occur unless the program adjusts itself when the decisions it makes are incorrect. The algorithm explained earlier changes the RULE array by adding the numbers in the VALUE array when an incorrect decision is associated with a negative value of TOT and subtracting the VALUE array when an incorrect decision is linked with a positive value for TOT. This is done by the function MODIFY. It is called by the main program when the user's response to the 'Is this correct?' prompt at the end of OPINION is to press the N key.

```
(DEFUN MODIFY NIL
 (SPACE)
 (FORMAT T "Rule needs modification")
 (SPACE)
 (FORMAT T "VALUE  RULE  RULE")
 (IF (< TOT 0) (FORMAT T " + ")
  (FORMAT T " - "))
 (FORMAT T " VALUE ")
 (DOTIMES (X N T)
```

```
(SPACE)
(BLANK 2)
(PRIN1 (AREF VALUE X))
(BLANK 5)
(PRIN1 (AREF RULE X))
(BLANK 9)
(IF (< TOT 0)
(SETF (AREF RULE X) (+ (AREF RULE X)
 (AREF VALUE X)))
(SETF (AREF RULE X) (- (AREF RULE X)
 (AREF VALUE X))))
(PRIN1 (AREF RULE X))))
```

First, FORMAT T is used to print the message 'Rule needs modification'. Then this heading is placed on the display if TOT is negative:

```
VALUE   RULE   RULE + VALUE
```

If TOT is not negative the display, instead, is:

```
VALUE   RULE   RULE - VALUE
```

The first part of this is done by (FORMAT T "VALUE RULE RULE"). A plus or minus sign will then be added, according to the value of TOT, by (IF (< TOT 0) (FORMAT T " + ") (FORMAT T " - ")). After this, the headings are completed by (FORMAT T " VALUE ").

A DOTIMES loop adds the entries beneath these headings. (BLANK 2) provides the initial indentation. The first entry under the heading VALUE is displayed by (PRIN1 (AREF VALUE X)). (BLANK 5) gives the next gap and the appropriate number from the RULE array is shown by (PRIN1 (AREF RULE X)).

Before the next entry is made, the modified array element in RULE, this array is updated. This adjustment is determined by TOT in the lengthy conditional expression:

```
(IF (< TOT 0) (SETF (AREF RULE X) (+ (AREF RULE X)
 (AREF VALUE X)))
  (SETF (AREF RULE X) (- (AREF RULE X)
   (AREF VALUE X))))
```

Upon analysis of the different levels of nesting, this can be seen to be embodying the algorithm which has already been explained. The new number in the RULE array is then added under the last heading by (PRIN1 (AREF RULE X)). (BLANK 9) gives the correct spacing.

The Learning System - ORACLE

All the component functions can now be assembled into the final function, ORACLE. The name is intended to boost the system's self-confidence!

```
(DEFUN ORACLE NIL
 (QUESTIONS)
 (FORMAT T "Test a subject?")
 (DO ((X 1 (1+ X)))
     ((NULL (Y-OR-N-P)))
     (ASK)
     (TABLE)
     (OPINION)
     (TERPRI)
     (IF (Y-OR-N-P) T (MODIFY))
     (TERPRI)
     (FORMAT T "Test a subject?"))
     (TERPRI)
 (FORMAT T "Exit from ORACLE"))
```

First, the function QUESTIONS is employed to establish the subject matter and the questions which the user will have to answer. A DO loop then follows. This will exit when the Y-OR-N-P predicate detects that the N key has been pressed. (NULL (Y-OR-N-P)) will then evaluate as T and the loop ends. A final message 'Exit from ORACLE' is displayed outside the loop.

Within the loop, the function ASK obtains the user's answers to all the questions. This information is then displayed by TABLE, as shown above. After this, OPINION makes and displays the system's choice and requests the user's confirmation.

After this is the expression (IF (Y-OR-N-P) T (MODIFY)). If the user does not agree with the decision made by ORACLE and presses N in reply to the prompt, (Y-OR-N-P) will have the value NIL and the function MODIFY is evaluated. This will adjust the RULE array accordingly.

The final prompt within the loop, displayed on the screen by (FORMAT T "Test a subject?") is linked with the exit condition at the beginning of the loop. For clarity of display, TERPRI needs to be used three times to space ORACLE's output neatly on the screen.

ORACLE will operate like this.

```
* (ORACLE)
Type first answer:
CAT
```

```
    Type second answer:
    DOG
    Type number of questions:
    3
    Type question No:1
    IS IT FURRY?
    Type question No:2
    DOES IT MIAOW?
    Type question No:3
    DOES IT BARK?
    Test a subject? (Y or N) Yes
    "IS IT FURRY?"
     (Y or N) Yes
    "DOES IT MIAOW?"
     (Y or N) No
    "DOES IT BARK?"
     (Y or N) Yes

REPLY  VALUE  RULE  RULE X VALUE

 YES     1     0         0

 NO      0     0         0

 YES     1     0         0

Total is 0 + 0 + 0 = 0
Positive value implies CAT
Is this correct?
 (Y or N) No

Rule needs modification

VALUE  RULE  RULE - VALUE

  1     0        -1

  0     0         0

  1     0        -1
Test a subject? (Y or N) Yes
"IS IT FURRY?"
 (Y or N) Yes
"DOES IT MIAOW?"
 (Y or N) Yes
"DOES IT BARK?"
 (Y or N) No
```

```
REPLY  VALUE  RULE  RULE X VALUE

 YES     1     -1        -1

 YES     1      0         0

 NO      0     -1         0

Total is -1 + 0 + 0 = -1
Negative value implies DOG
Is this correct?
 (Y or N) No
Rule needs modification

VALUE  RULE  RULE + VALUE

  1     -1         0

  1      0         1

  0     -1        -1

Test a subject? (Y or N) Yes
"IS IT FURRY?"
 (Y or N) Yes
"DOES IT MIAOW?"
 (Y or N) No
"DOES IT BARK?"
 (Y or N) Yes

REPLY  VALUE  RULE  RULE X VALUE

 YES     1     0         0

 NO      0     1         0

 YES     1    -1        -1

Total is 0 + 0 + -1 = -1
Negative value implies DOG
Is this correct?
 (Y or N) Yes
Test a subject? (Y or N) Yes
"IS IT FURRY?"
 (Y or N) Yes
"DOES IT MIAOW?"
 (Y or N) Yes
"DOES IT BARK?"
 (Y or N) No
```

```
REPLY   VALUE   RULE   RULE X VALUE

 YES      1      0          0

 YES      1      1          1

 NO       0     -1          0
Total is 0 + 1 + 0 = 1

Positive value implies CAT

Is this correct?
 (Y or N) Yes
Test a subject? (Y or N) No
Exit from ORACLE
NIL
```

An Astronomical Application

ORACLE can now be used in a second example of machine learning in which it will again be seen building up a rule array capable of distinguishing between two subjects. This time it has to decide whether a particular member of the Solar System is the Moon or the planet Mars. Six questions are asked and ORACLE has to develop array values which will allow it to decide which of the two possible answers is the appropriate planetary body, according to the responses typed to these questions.

The questions, together with the relevant replies, are as follows:

Questions	Moon	Mars
1. DOES IT ORBIT EARTH?	YES	NO
2. IS IT SMALLER THAN EARTH?	YES	YES
3. IS ITS SURFACE CRATERED?	YES	YES
4. DOES IT HAVE SATELLITES?	NO	YES
5. HAVE ASTRONAUTS LANDED THERE?	YES	NO
6. ARE THERE POLAR CAPS?	NO	YES

Note that again these questions divide into three categories, which will have correspondingly distinctive values in the rule array.

Questions 2 and 3 both give the same response for Moon and Mars. Both the Moon and Mars are smaller than the Earth and possess heavily cratered surfaces. This is probably because neither body has ever had a thick atmosphere to cause erosion of the original surface, as was apparently the case with the Earth. The answers to these two questions do not therefore make any distinction between the two possible members of the Solar System. This means that the questions will not contribute to the rule array. The numerical values in the final array will necessarily be zero.

Questions 1 and 5 both have a positive reply for the Moon and a negative one for Mars. The Moon does orbit the Earth and has been visited by the astronauts of Project Apollo. The rule array will need to contain a 1 in the first and fifth array locations in order to produce a positive value for the variable TOT and so select the first subject, the Moon.

Conversely, questions 4 and 6 have negative responses for the Moon and positive for Mars. Mars has two tiny satellites of its own, Phobos and Deimos, and has enough of an atmosphere to allow gases to condense into polar caps, as happens on the Earth. The rule array will need to have a value of -1 in its fourth and sixth locations in order to lead to a negative value for TOT and select Mars as ORACLE's deduction.

The finished rule which is eventually developed by ORACLE should therefore be (1 0 0 -1 1 -1). When this rule has been achieved, correct decisions will inevitably be made when answers are typed in for the six questions that the program asks. Of course, the rule begins simply as (0 0 0 0 0 0) and ORACLE needs to fill in the correct array values as its initial decisions are either confirmed or rejected by the user.

Here, then, is ORACLE dealing with this situation.

```
* ORACLE
Type first answer:
MOON
Type second answer:
MARS
Type number of questions
6
Type question No:1
DOES IT ORBIT EARTH?
Type question No:2
IS IT SMALLER THAN EARTH?
Type question No:3
IS ITS SURFACE CRATERED?
Type question No:4
DOES IT HAVE SATELLITES?
Type question No:5
HAVE ASTRONAUTS LANDED THERE?
Type question No:6
ARE THERE POLAR CAPS?
Test a subject? (Y or N) Yes
"DOES IT ORBIT EARTH?"
 (Y or N) No
"IS IT SMALLER THAN EARTH?"
 (Y or N) Yes
"DOES IT HAVE CRATERS?"
 (Y or N) Yes
"DOES IT HAVE SATELLITES?"
 (Y or N) Yes
```

```
"HAVE ASTRONAUTS LANDED THERE?"
 (Y or N) No
"ARE THERE POLAR CAPS?"
 (Y or N) Yes
REPLY VALUE  RULE  RULE X VALUE
 NO      0     0          0

 YES     1     0          0

 YES     1     0          0

 YES     1     0          0

 NO      0     0          0

 YES     1     0          0

Total is 0 + 0 + 0 + 0 + 0 + 0 = 0
Positive value implies MOON
Is this correct?
 (Y or N) No
Rule needs modification

VALUE  RULE  RULE - VALUE
  0     0          0

  1     0         -1

  1     0         -1

  1     0         -1

  0     0          0

  1     0         -1

Test a subject? (Y or N) Yes
"DOES IT ORBIT EARTH?"
 (Y or N) Yes
"IS IT SMALLER THAN EARTH?"
 (Y or N) Yes
"DOES IT HAVE CRATERS?"
 (Y or N) Yes
"DOES IT HAVE SATELLITES?"
 (Y or N) No
"HAVE ASTRONAUTS LANDED THERE?"
 (Y or N) Yes
"ARE THERE POLAR CAPS?"
 (Y or N) No
```

```
REPLY  VALUE  RULE  RULE X VALUE
 YES     1      0         0

 YES     1     -1        -1

 YES     1     -1        -1

 NO      0     -1         0

 YES     1      0         0

 NO      0     -1         0

Total is 0 + -1 + -1 + 0 + 0 + 0 = -2
Negative value implies MARS
Is this correct?
 (Y or N) No
Rule needs modification

VALUE  RULE  RULE + VALUE
   1     0         1

   1    -1         0

   1    -1         0

   0    -1        -1

   1     0         1

   0    -1        -1
Test a subject? (Y or N) Yes
"DOES IT ORBIT EARTH?"
 (Y or N) No
"IS IT SMALLER THAN EARTH?"
 (Y or N) Yes
"DOES IT HAVE CRATERS?"
    (Y or N) Yes
   "DOES IT HAVE SATELLITES?"
    (Y or N) Yes
   "HAVE ASTRONAUTS LANDED THERE?"
    (Y or N) No
   "ARE THERE POLAR CAPS?"
    (Y or N) Yes

   REPLY  VALUE  RULE  RULE X VALUE

    NO      0     1         0

    YES     1     0         0
```

```
     YES      1      0          0

     YES      1     -1         -1

     NO       0      1          0

     YES      1     -1         -1

    Total is 0 + 0 + 0 + -1 + 0 + -1 = -2
    Negative value implies MARS
    Is this correct?
     (Y or N) Yes
    Test a subject? (Y or N) Yes
    "DOES IT ORBIT EARTH?"
     (Y or N) Yes
    "IS IT SMALLER THAN EARTH?"
     (Y or N) Yes
    "DOES IT HAVE CRATERS?"
     (Y or N) Yes
    "DOES IT HAVE SATELLITES?"
     (Y or N) No
    "HAVE ASTRONAUTS LANDED THERE?"
     (Y or N) Yes
    "ARE THERE POLAR CAPS?"
     (Y or N) No

REPLY  VALUE  RULE  RULE X VALUE
 YES      1     1          1

 YES      1     0          0

 YES      1     0          0

 NO       0    -1          0

 YES      1     1          1

 NO       0    -1          0

Total is 1 + 0 + 0 + 0 + 1 + 0 = 2
Positive value implies MOON
Is this correct?
 (Y or N) Yes
Test a subject? (Y or N) No
Exit from ORACLE
NIL
```

Summary

1. Expert systems are a significant part of the A.I. revolution. They represent a genuine application of A.I. in the real world.

2. Knowledge can be represented in an expert system as frames, scripts, semantic networks or production rules.

3. The inference engine uses the knowledge-base in its decision making.

4. Machine learning will eventually make the construction of expert systems far easier.

5. Feedback and 'learning by small amounts' are important aspects of machine learning.

6. It is unnecessary to attempt to simulate 'true' understanding.

7. `READ-LINE` allows a whole sentence to be entered into Golden Common LISP.

Appendix A

Some A.I. Themes

Artificial Intelligence

This is a branch of computer science which explores the possibility of creating computer software capable of mimicking some of the abilities of the human mind. The name Artificial Intelligence, usually shortened to A.I., dates from the 1956 Dartmouth conference, where it was coined by John McCarthy. In the years since then continual progress has been made towards the initial objectives set by A.I. researchers, often following specific insights, like the rapid development of expert systems. This resulted from the realisation that abstract reasoning mechanisms were less likely to succeed than software which contained explicit information relevant to the area in which the program was intended to operate. In contrast, earlier A.I. workers had hoped to create general systems capable of logical problem-solving over broad areas of application.

A.I. research is directed towards a number of specific applications. Many of these have a commercial orientation. Others are more theoretical, intended to illuminate some aspect of human intellectual ability and to recreate this in a computer. Natural language understanding is an important part of A.I. research and similarly computer vision. Expert systems and robotics also figure largely in the field. Currently, the most widely used programming language in A.I. is still LISP, although the more recently developed PROLOG has rapidly established itself in second place.

The main thrust of A.I. research is now probably determined by the fifth generation programme initiated in Japan. Ultimate advances in A.I. will depend largely upon two factors, the development of new computer architectures incorporating parallel processing and the creation of more sophisticated languages capable of exploiting the new possibilities of such enhanced machines. Combined with the continuing growth of programming experience and techniques, these will lead to greater insights into the nature of mind as well as practically useful robots and intelligent software.

Blocks World

In his natural language program SHRDLU, Terry Winogram limited the domain of discourse to a simplified microworld made up of imaginary geometrical objects, blocks, cubes and pyramids, and a manipulative robot arm. This reduction of the world's complexity made it possible to create a high level of linguistic comprehension in the program's description of what was taking place.

Boden, Margaret

Reader in Philosophy and Pychology at the University of Sussex, Margaret Boden is concerned with the philosophical and social consequences of A.I. and in particular the possible rights of sentient machines. She is the author of *'Artificial Intelligence and Natural Man'* (Harvester Press, 1977).

Boltzmann Architecture

In a hypothetical computer design involving parallel processing and intended to simulate the functioning of the human brain, a large number of separate processors are placed initially in a random state and allowed to coalesce in the hope that their new relationship will mimic the brain's power of intuition.

Capek, Karel

The term 'robot' has become so ubiquitous that it is easy to ignore its relatively recent origin. It dates from 1921, when the play *'Rossum's Universal Robots'* was first performed in Czechoslovakia. At first principally referring to slave labour, the word soon came to be identified with mechanical men. The play was written by Karel Capek, together with his brother Josef. Capek had a resigned view of mankind's future and feared that science would not be controlled. He wrote about nuclear war in *'The Absolute at Large'* (1922) and a John Wyndham-like fantasy about strange sea creatures drowning the continents in *'The War with the Newts'* (1936). Despite his relative obscurity, Capek has provided a remarkably varied stimulus to 20th Century thought. In addition to inventing robots, his *'The Markropoulos Secret'* (1927), in which a formula for immortality is discovered, was transformed into an opera by Janacek and subsequently a philosophical debate by Bernard Williams, published in his *'Problems of the Self'* (Cambridge University Press, 1973).

Cognitive Science

This is a new discipline drawing upon computer science, linguistics and psychology in order to construct models of the mind capable of explaining the processes involved in intelligent thought. The amalgamation of these separate areas of interest into cognitive science has occurred quite naturally as common issues for investigation have become apparent. For example, the way in which language is understood and used creatively is of concern to the computer scientist who wants to develop a natural language front-end for a piece of intelligent software, as well as for the linguist exploring the subtleties of syntactic construction, or for the psychologist investigating the way in which language is acquired by the growing infant. Similarly, perception, knowledge representation and problem-solving are some of the shared topics for all three sciences.

Cognitive science helps to unite the different disciplines by adopting a common set of methods for their mutual research. This includes heuristic search, semantic nets, production systems and many other theoretical tools drawn from Artificial Intelligence. In fact, one of the objectives of cognitive scientists is to be able to take concepts from A.I. and use them to help explain the way in which people think. As a corollary of this, an ideal vision of the state of cognitive science would involve psychological theories in the construction of more sophisticated A.I. programs.

Commonsense

This is the human ability (and current computer inability) to cope with the complex, though apparently trivial, activities of daily life. In the early history of A.I. research, computer scientists greatly underestimated the difficulty of giving a program common sense reasoning. This led to an unjustified optimism about the speed with which significantly intelligent machines could be developed.

Computer Creativity

The supposed ability of a machine to generate seemingly original ideas could be termed computer creativity. It is fairly easy to simulate apparent artistic ability with the even with the simple programs capable of being run on small personal computers:

```
ODE TO LUCY

THE DEVOUT LAKE REMINDS
ME OF YOUR CRY,
EVEN THOUGH YOUR WAIST
IS RATHER NIGH.
THE MAGICAL MOUNTAIN REMINDS
ME OF YOUR REAR,
EVEN THOUGH YOUR SPITE
IS RATHER NEAR.
MY LUCY, LUCY, LUCY,
SO STRANGELY LACED,
NO WONDER SINCE YOU'VE GOT
A VERY ICY BASE.
THE PURPLE FIRE COULD NEVER
BE SO CHASTE,
AS LUCY IN THE MORNING
WHEN YOU CHASE.
```

Here, the reference to the major novel by Thomas Mann, and the subtle congruence between 'mountain' and 'icy base' were both wholly accidental. They were simply a consequence of leaving the personal computer (in this case a Sinclair QL) running long enough and discarding most of the print out. Rather alarmingly, even large main frame computers running 'creative' programs often appear to produce similarly incoherent free associations. A cynic might suppose that computers might eventually be capable of Dylan Thomas, but never T.S.Eliot... However, some computer programs have displayed remarkable originality in other areas, a notable case being Lenat's EURISKO.

Computer Vision

An important area of research in A.I. is the development of computer systems which are able to process the digital information provided by a video camera in a way which creates a realistic internal model of the scene involved. Ideally, this should be of a sufficient degree of complexity and intelligence to allow the system to interact practically with the environment. This, for example, could involve the control of a robot arm adding or removing components on a factory conveyor belt, or even the navigation of a vehicle through terrain strewn with various types of obstacle. The practical problems are considerable, as witnessed by the high proportion of the human cortex devoted to visual processing.

The first stage in the analysis of an image is its reduction to an array of pixels recording various levels of grey. The additional visual cues provided for the human by colour perception are necessarily ignored by current systems. Various algorithms are

then involved in the detection of lines and edges within the pixel array. This is one of the essential features of the scene to be determined and much research has been conducted into basic methods for the resolution of raw information into a set of cartoon like lines. After this, the vertices formed by the set of lines need to be identified in a consistent, unambiguous way which avoids the spurious identification of impossible shapes. Matching can then take place with what the program would expect to see, working from the visual information held within a data base. Although the complexity of the programming involved is daunting, computer vision is one of the areas of A.I. where research is proceeding quite rapidly, probably as a result of the considerable incentive of enhanced capabilities for industrial robots.

Dreyfus, Hubert

A Professor of Philosophy at the University of California, Dreyfus is one of the opponents of the concept of 'strong' A.I. and maintains that computers are not capable of genuine thought. He believes that in the history of Western philosophy from the time of Plato onwards there has been an excessive distinction between the mind and the surrounding world. This overlooks the important contribution made to the personality by emotions, perceptions and the general interaction of self and the cosmos. It also leads, in his opinion, to the implicit, though false, assumption made by proponents of strong A.I. that the mind can be codified into a formal system of rules which can then be instantiated by a sophisticated computer program. He is the author of *'What Computers Can't Do: The Limits of Artificial Intelligence'* (New York: Harper & Row, 1979).

ELIZA

Joseph Weizenbaum wrote this program at M.I.T. in 1966, intending it to simulate intelligent dialogue. It plays the role of a doctor offering pyschiatric counselling to the person interacting with it. It has no genuine understanding of natural language, but instead proceeds by a process of 'cocktail conversation', giving prepared answers for certain key words and phrases when it detects them and responding with sufficient vagueness otherwise. Nevertheless, Weizenbaum was horrified when he discovered how readily some people were deceived by the program, believing that they were genuinely in remote conversation with a skilled psychiatrist. This has some bearing upon the question of whether the Turing test is a useful measure of intelligent behaviour.

EURISKO

A program developed by Douglas Lenat at Stanford University, EURISKO has managed to create a far more respectable image for computer creativity than many other previous attempts. Derived from Lenat's earlier work on Automatic Mathematician, it employs hundreds of generalised heuristic rules to allow it to tackle a wide range of problems. In order to mimic the way in which people seem to learn through discovery and creativity, Lenat deliberately allows EURISKO to modify these heuristcs for itself. He has discovered that this leads to an enhanced mode of operation for the algorithms employed. The program apparently displays genuine creativity. Applied to the design of very large scale integrated circuitry, EURISKO devised a chip architecture which actually proved sufficiently novel and useful to be patented. It later

applied some of the heuristics which it had evolved in its experience of V.L.S.I. chip design to a naval war game called Traveller T.C.S. Here EURISKO won a simulated encounter between two fleets by selecting the unexpected strategy of fighting with a fleet of small, fast vessels. Eurisko's behaviour is goal directed and it constantly reviews its performance on a 'scale of worth'. Lenat has found, to his dismay, that it regularly rediscovers a new heuristic which assigns high scores (which he immediately deletes). This is simply to attach its own name as creator to any recent discovery of high worth. Lenat has found it extremely difficult to teach EURISKO not to cheat!

Expert System

Expert systems are a major part of A.I. programing, offering the user all the advantages of advice from a specialist but with the convenience of a computer program that can be operated whenever necessary. Unlike human experts, expert systems do not get cross, become tired or confused, or die. A typical system consists of a natural language front-end, the inference engine, which carries out the automatic reasoning, and a detailed knowledge-base where all the relevant information is stored. Various expert systems have become extremely well known, like MYCIN, DENDRAL and PROSPECTOR and the development of expert systems has increased interest in A.I. research. Expert systems do have disadvantages as well. The creation of a good system takes a great deal of time and money, and even then it does not necessarily lead to correct answers the whole time. It is rather the case of sensible and appropriate advice being offered for the careful consideration of a human decision maker. A further drawback to the application of expert systems is that they do require large machines on which to operate. Of course, with the continuing improvement of small computers this will gradually cease to be a major problem.

Fifth Generation

This is the proposed next stage of computer development. Launched in October 1981 and particularly associated with Japan, it involves sophisticated A.I. in the design of more advanced hardware and software. Fifth generation computers are intended to communicate in natural language, possess a large knowledge-base, together with an efficient inference engine, and be able to derive new heuristics from their own experience.

Knowledge-base

The information used by an expert system is referred to as its knowledge-base. A number of different types of knowledge representation are available for the knowledge-base.

Knowledge Engineering

When an expert system is established, information has to be transferred from human experts in the chosen area into the knowledge-base in the computer software. The knowledge has to be stored in a way suitable for the operation of the expert system. Translating the specialist's expertise into the knowledge-base is described as knowledge engineering.

Knowledge Representation

A problem encountered in expert systems, and other software in which a particular body of information is required, is the choice of a suitable method for storing the knowledge involved. The way in which the program will employ the information will usually make some methods of storing it, the knowledge representation, more suitable than others. Various types of knowledge representation have been devised, for example, frames, scripts, networks and production rules.

Lambda Calculus

The logician Alonzo Church introduced a new notation, the lambda calculus, to help distinguish in formal logic between a function and the values which is capable of assuming. In this, a logical operator, represented by the Greek letter lambda, was 'abstracted' from the function and known as an abstraction operator. Some of the structure of Church's calculus was carried over into LISP.

LISP

This language was first devised by John McCarthy in 1958 and has since become the major language for A.I. applications. Since its original conception, a number of dialects of LISP have been developed. Two of these are MacLISP, created at the Massachusetts Institute of Technology, and InterLISP, devised at Palo Alto. Both have been used extensively on dedicated LISP machines. Other dialects include ZetaLISP and le LISP. Common LISP arose in the USA from a deliberate effort to produce a definitive standard for the language, consultations for which were instigated by the Defence Advanced Research Projects Agency in 1982. A design team was set up including experts from various industries and universities, Yale, Carnegie-Mellon, Digital Equipment Corporation, Bell Laboratories and many other similar organisations. Golden Common LISP is an implementation of Common LISP developed by Gold Hill Computers for the PC market. Unlike some of the other dialects of LISP available for microcomputers, it allows genuine program development rather than being limited by memory restrictions to the role of a 'toy' language.

LISP Machine

LISP is a language which requires a large amount of available memory. As a result of this, many of the versions operating on small personal computers are perhaps little more than 'toy' dialects, useful for learning the language but inappropriate for significant applications. The essential structure of LISP inevitably means that the amount of memory employed in moderately ambitious programs can even exceed that available on big machines. A way of tackling this problem was devised at M.I.T. with the concept of the 'LISP machine', a computer dedicated to LISP programming and possessing a number of features designed to ease the memory restriction. A typical LISP machine is likely to possess inbuilt LISP primitives, up to 100 megabyte of available memory, large word size, efficient garbage collection, a very high resolution monitor and the ability to upgrade the system as necessary.

LOGO

Seymour Papert developed a further list processing language from LISP and named it LOGO. It is a powerful language and capable of serious applications. However, it is usually perceived in the special role Papert selected, that of an extremely suitable

language for school and educational use. Working with a minimal core of primitive commands, even young children are able to build up their own personal library of procedures. Naturally, this creates a feeling of confidence and familiarity with computers as a tool which would be unlikely to occur with other programming languages. LOGO can create intricate graphic displays by directing a screen 'turtle' and is also able to direct the movement of small robot-like turtle vehicles. A typical example of a simple procedure a child might write would be as follows:

```
BUILD SQUARE
REPEAT 4
FORWARD 100
LEFT 90
END
```

Here, a loop has been used to construct the four sides of a square and the procedure given the obvious name. Once this definition has been entered into LOGO, the command SQUARE is automatically added to the available primitives and can then be called whenever a square is to be drawn on the monitor or traced by the floor turtle.

Machine Learning

During the lobotomy performed on the crazed HAL in Arthur Clarke's '*2001 - A Space Odyssey*', the machine is seen regressing to a computational infancy. Here, in the film's major moment of pathos, a disembodied voice haltingly sings 'Daisy, Daisy...' before sinking into intellectual oblivion. Arthur Clarke drew heavily at this point upon some of the early designs for machine learning, which were derived from the belief that the ideal way of allowing a computer to acquire knowledge is to model it in some fashion upon the human infant. The developing human personality gradually acquires knowledge of the world through the senses, combined with the mind's ability to reshape what it needs to build up a coherent account of everything. In like fashion, the computer of the future was intended to be encouraged to find out things for itself. Pre-Dartmouth Conference work in cybernetics experimented in this direction, attempting to build electronic brains which directly copied some aspect of what was then known about the organisation of the human cortex. The Perceptron designed by Rosenblatt was a practical example of this. Related theory was developed by the neuro-physiologist, Warren McCulloch, with mathematician colleague, Walter Pitts, and also by W.R. Ashby, a neurologist friend of Alan Turing. As a direction for future research, however, this approach proved relatively fruitless.

Instead an alternative model of the human mind became emphasised, which proposed that no growth of learning ever really begins from a mental ground zero. New information is always fitted into some existing framework. The assumption that people are born with useful knowledge already hard-wired into the brain has a long history. The concept appears in Plato's belief in innate ideas and carries on through the history of philosophy, via figures like Descartes. In modern times it appears in the transformational grammar proposed by Naom Chomsky.

More recent A.I. programs designed to acquire knowledge therefore slot new information into existing schemata. According to Eugene Charniak and Drew McDermott, 'for an organism to learn anything, it must already know a lot. Learning begins with organised knowledge, which grows and becomes better organised', (*'Introduction to Artificial Intelligence'*, (Addison-Wesley, 1985), p.610). This approach is realised in two of the principal components of any machine learning system. One is the need for learning by small amounts that can be reconciled with what is already in place. The other is the importance of feedback in checking the success of new predictions against what has been established in the knowledge-base.

McCarthy, John

One of the computer scientists who must be regarded as having played a leading role in the creation of artificial intelligence as a new, distinct discipline is John McCarthy. It was he who, with the help of Marvin Minsky, organised the 1956 conference at Dartmouth College, New Hampshire, which is now generally recognised as having marked the official birth of A.I. In fact, this conference saw the first formal use of the term 'Artificial Intelligence', also invented by McCarthy, who had rejected his earlier, provisional name 'automata studies' as being too imprecise. Additionally, McCarthy created the language of LISP itself in 1958, when he was Assistant Professor of Mathematics at Dartmouth College. He has also founded two major laboratories of A.I. research, at M.I.T. in 1957 and at Stanford, where he is currently Professor of Computer Science and Director of the A.I. Laboratory, in 1963.

Minsky, Marvin

One of the original four founding figures of A.I., Marvin Minsky has continued at the forefront of research ever since. He created the frame approach to knowledge representation. Minsky is a notable proponent of strong A.I., suggesting that eventually it may even be feasible to shift human personalities into machinery and away from the 'billion-year accumulated mess' of organic bodies.

Parallel Processing

Traditional computer architectures segregate the single processing unit from its associated memory. Data from this has to be retrieved sequentially as the machine operates. The resultant linear processing inevitably places a limit upon the speed and power of the computer. The reason for adopting such a configuration is largely accidental. It is a consequence of the historically different physical structures of the central processor, originally electronic valves, and the memory, then a physically incompatible hybrid of mechanical and magnetic devices. With modern techniques of construction this division is no longer required. It is, in fact, a hinderance to faster and more efficient computation. Instead, it is more convenient to combine processing and memory on to the same silicon chips, a large number of which are then combined to create a computer which processes information, not sequentially but simultaneously or in parallel.

A practical example of this application of parallel processing to computer architecture is the Connection Machine, an advanced computer manufactured by Thinking Machines Corporation. It is remarkably unlike traditional computers in appearance,

consisting of eight cubes assembled into one larger cube, 1.5 metres high. Each subcube holds 16 vertical boards. Every board has 32 chips consisting of 16 processors. The total number of these is therefore 8 X 16 X 32 X 16, or 65 536. In order to simplify maintenance of the computer, each of the 4 096 chips is connected to a red light on the outside of the subcube, the pattern formed during its operation making a Connection Machine appear to have been beamed down directly from Captain Kirk's Enterprise.

Another machine using parallel processing is the Meko Computing Surface. This is based upon the transputer, a radically different device developed at Inmos and combining processing and memory on a single chip. Transputers are designed to be able to communicate easily with one another and so it is not difficult to have large numbers linked together for parallel processing. The Computing Surface currently employs 300 transputers, but an enhanced machine incorporating 1024 transputers is being built for the University of Edinburgh.

A problem created by computers like these is the importance of connecting the separate processors as directly as possible. Many of the advantages of the parallel architecture would be lost if the network connecting the processors were inefficient in design. One solution is to base the network upon n-dimensional hypercubes. A two-dimensional square connects four corners with no pair of corners more than two sides away from each other. A three-dimensional cube has eight vertices separated at most by three edges. A four-dimensional hypercube, formed by joining the vertices of two cubes, has 16 vertices never separated by more than four arcs. The pattern continues into higher dimensions and so is ideal for limiting the separation of large numbers of processors.

A further complication with these new machines is the need for more advanced programming languages, a typical example being the development of Occam.

Parsing

One of the stages involved in the analysis of natural language is parsing, or the division of the words of a sentence into their respective roles. This helps to make the grammatical structure of the sentence explicit. As an example, the sentence 'The green frog croaked excitedly' would lead to a 'parse tree' as shown on the next page:

Splitting up sentences in this fashion is an essential early stage of language analysis, but it is still very far from permitting genuine machine understanding of the meaning of the sentence. One difficulty is that structural identity of different sentences does not necessarily imply similar semantic content. Apochryphal stories abound about early translation programs which could equate 'Out of sight, out of mind' with 'Invisible idiot'. One example given by an A.I. researcher of the difficulty of creating machine understanding was the contrast in meaning between 'Time flies like an arrow' and 'Fruit flies like a banana'.

A program designed to parse a sentence has to identify a correct part of speech for each word so that a unambiguous sentence structure eventually results. The most favoured method for doing this, devised independently by Woods and Dewar in 1969-70, is to

construct an augmented transition network, or ATN, in which the nodes represent possible parsings of parts of the sentence and the arcs are tests deciding whether a consistent analysis is being created.

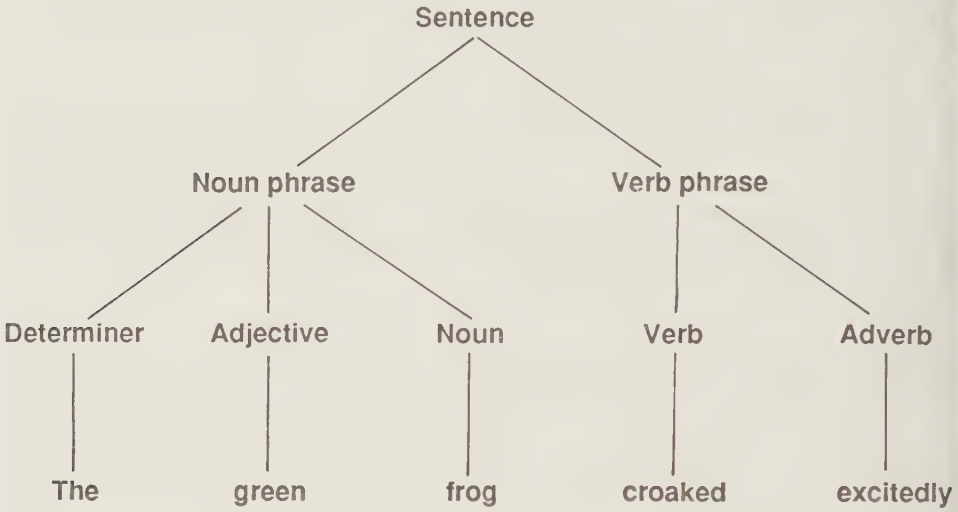


Figure A.1

Pattern Matching

Pattern matching is important in LISP and A.I. and occurs in varied applications. Part of the processing of a visual image will involve checking the arrangement of lines and vertices against previously learnt shapes. A natural language sentence, after parsing, may be compared in its structure with past examples. In both cases, a fundamental role is played by pattern recognition. Fortunately, it is relatively easy to construct pattern matching functions in LISP.

Philosophical Issues in A.I.

A significant characteristic of artificial intelligence research is the way in which it overflows into other areas of intellectual inquiry and sheds a new light upon many traditional issues. A typical example of this is the new perspective it brings to the philosophical debate about the relationship of the human mind to the human body. Most people probably reflect occasionally on the sheer oddness of human sentience. From time to time it is impossible not to wonder about the strange way in which one is distinct from everything else, from the surrounding world and from other human beings. This feeling has given rise to many disparate reactions. At one extreme it can encourage religious fervour. Taken to the other, it may create suicidal depression. It is a contributary cause of the alienation catalogued in 20th century French literature by Sartre and Camus. Meanwhile psychological research has attempted to explain the relation between body and mind and A.I. becomes involved in two separate ways. First, its typical concepts can be applied profitably to the question of human consciousness

itself, introducing ideas which supplement conventional philosophical argument. At the same time, the possibility of ultimate machine consciousness creates further problems requiring the help of the philosopher in their resolution. Thus A.I. and the philosophy of mind can mutually interact, each providing a stimulus for the other.

In order to help formalise the problems involved in coming to an understanding of the mind, philosophers identify two major components of the mental which can contribute towards a formal definition. One is the existence of consciousness or self-awareness. The other is the notion on 'intentionality' or directedness. When we believe, we believe in something. When we worry, we worry about something. We do not just believe or worry. Our mental states are directed towards some object of our thought. There is a significant difference here with, for example, kicking. When you kick, you kick something, a ball, your car or a rival suitor. But these objects are physical presences in the world. The odd thing about intentional mental states is that the objects to which they are directed need have no real existence at all. You might believe in Nessie, flying saucers or the Great Pumpkin. Intentionality has reasserted itself in recent years as a major problem for proponents of strong A.I. and has been used as a weapon against it by philosophers like John Searle.

A contrast between matter and ideas is found in the earliest Western philosophy, but conventionally the mind-body problem is regarded as first appearing as a specifically philosophical issue in the work of the great French philosopher and mathematician, Rene Descartes (1591-1650). His encompassing philosophical system stemmed from his initial bulwalk against total scepticism, the conviction that he must really exist since he was aware of his thoughts. This led directly to his imposed distinction between the mental and the physical and the subsequent problem in reconciling the two. Naturally, Descartes was aware that his mind appeared to have direct powers over his body. Unable to explain this to his satisfaction, he eventually grasped almost in desperation at the pineal gland as the neutral territory between mental and physical substances. However, he left as part of his legacy a philosophical issue which has remained contentious. Many philosophers following Descartes have also proposed similar dualist theories in which the mental and physical are kept separate.

In contrast, monist theories of the mind reject the existence of opposing mental and physical substances and suppose that there is only one underlying explanation accounting for both. Spinoza (1632-77) believed that mind and matter were simply different aspects of an infinite all-pervading substance which he equated with God. Idealist philosophers like Berkeley (1685-1753) and Hegel (1770-1831) regarded mind as the fundamental reality. One- substance philosophies now place supremacy upon the physical world. Central-state materialism is a typical example of a contemporary theory which identifies emotions, beliefs, intentions and other mental events directly with neuro-physiological states of the brain.

A typical problem encountered in the philosophy of the mind is illustrated in the work of David Hume (1711-76). He accepted the Cartesian distinction between human consciousness and the external material world and investigated what he regarded as

'perceptions of the mind'. These he divided into primary impressions resulting from the senses and ideas derived from the impressions. He believed that these mental perceptions were what we were really conscious of, rather than the world itself. But here he encountered a problem which he was unable to solve. It seemed to him that his model of the mind required some kind of internal observer which would be aware of the perceptions and thus create consciousness. However, this homunculus, or little man perched mysteriously within the human mind simply placed the whole problem at a deeper level of analysis. It postponed the issue rather than solving it and, taken seriously, could only lead to an infinite regress with another homunculus placed within the first. Hume's problem has remained taxing ever since.

A new insight into this dilemma is given by a modern philosophical approach to the mind-body problem. This is the functionalist model of the mind, inspired by research in cognitive science and A.I. and made explicit in, for example, the work of Jerry Fodor and Daniel Dennett. It is a materialist theory of the mind and does not assume the existence of any mysterious spirit to explain consciousness. At the same time it avoids simply equating states of mind with physical states of the brain, as suggested by central state theories. Instead mental states are linked with the functional states of the brain. Such an explanation is not really concerned with a bottom-up approach to the problem of consciousness, in terms of neurological structure of the cortex. Rather a top-down strategy is adopted and the operation of the brain is viewed as questions of the organisation, logical operations and rules of transformation required to lead from one functional state to another. A consequence of this model, leading to philosophical criticism of it, is that really the biological brain is no longer needed as the necessary functional states can equally well be realised by sufficiently complex electronics. Strong A.I. seems an inevitable consequence of functionalist theories.

Applying the functionalist method, Dennett tackles Hume's problem in his *'Brainstorms: Philosophical Essays on Mind and Psychology'* (Harvester Press, 1978). Instead of having a single homunculus doing all the necessary observation needed to create the totality of a conscious human mind, he supposes that an intelligent system, whether modelled on a computer or actually occurring in a real person, could be created as a series of sub-systems. Each has a particular role to play and contributes its own little piece to the overall mind. The single homunculus then becomes a team and thus the infinite regress is avoided. However, Dennett's position has been criticised by some philosophers who feel that his reduction of the mind to a team of near-moronic homunculi avoids the true problem encountered by Hume, which instead is really one of intentionality and not removed simply by reducing the scale of application.

Additionally, Dennett's account of the human mind may seem an unsatisfying explanation for the average layman. There is an understandable urge to single out one of the sub-systems as the president elect amongst the homunculi and to identify that as being truly the self. However, this would again appear to lead back again to the regress Hume sought to avoid. A further complication is also added by current research in cognitive science. This implies that if any sub-system is to be identified with what apparently is conscious in the human mind it is not some overall governing entity.

Instead it would be a relatively low level unit with limited access to what is really taking place in our mental life. The evidence for this seems quite compelling. In a rather disturbing sense, we become passengers within our own selves, rather than the all-seeing driver, and are literally strangers to our minds.

A further philosophical contribution to A.I. is seen in the new recognition being given to the work of Edmund Husserl (1859- 1938), now regarded as being of significance for contemporary research. Husserl developed further the philosophical doctrine of phenomenalism which he inherited from his tutor, Franz Brentano. He sought for a closer identification of things perceived and the actual perception of them, linked by the intentionality of consciousness. The contemporary American philosopher Hubert Dreyfus has even described Husserl as the father of artificial intelligence because his belief that the mind follows a hierarchy of rules is of immediate applicability in the construction of machine intelligence. Nevertheless, this opinion probably has to be qualified by a critical appraisal of the status of strong A.I. which Dreyfus has also given elsewhere.

PROLOG

Currently LISP is the most popular programming language employed in creating A.I. software and has a clear lead over any of its rivals. It has benefitted from being the first computer language intended specifically for A.I. applications and the important developments in the field have mainly been achieved using LISP. However, it is no longer the only contender for A.I. programming. A significant new language has been created for A.I. applications, PROLOG, or PROgramming in LOGic. It was developed in France by Alain Colmerauer and based upon predicate calculus. PROLOG has established a interesting new European slant to A.I. research. The language has undergone considerable enhancement, particularly in the U.K. As a result of its rapid evolution no definitive standard yet exists for PROLOG, unlike the uniformity conferred upon a multitude of LISP dialects by the creation of Common LISP. The language has, however, been selected as the main choice for the Japanese fifth generation programme.

Putnam, Hilary

Another contemporary philosopher who has become interested in issues raised by A.I., Putnam is author of *'Robots: machines or artificially created life?'* and 'The mental life of some machines', published in his *'Mind, Language and Reality'* (C.U.P. 1975). He is professor of both philosophy and of mathematical logic at Harvard University.

Raibert, Marc

One of the practical problems which needs to be solved before true, human-like robots are generally admitted to society is the creation of genuine walking machinery. Despite the animated oil drum model cheerfully currently accepted by the film industry, the ideal domestic robotic should not be allowed simply to trundle around on wheels or caterpillar tracks. Instead, it ought to be capable of sitting down occasionally to tell stories to the children and be able to climb the stairs to put out clean towels in the guest room. Fortunately, much work has been carried out by Marc Raibert, Professor of Computer Science and Robotics at Carnegie-Mellon University, into the practical

difficulties of designing machines that mimic the way animals and human beings walk. This has been shown to require considerable computational power. Nevertheless, remarkable success has been achieved in producing legs that walk and run, partially as a result of initial concentration upon the software required for the balance embodied in a single, hopping leg. Bipedal locomotion, an essential component of the true future robot, can be considered now well within reach.

Robotics

This has tended to become an extremely nebulous term, applied quite happily to a variety of activities ranging from the science fiction writing of Isaac Asimov to a briefly popular teenage dance craze in the mid-1980s. A tighter definition would have to include two main components, accurate and sophisticated mechanical engineering used to create a very high level of control and the application of A.I. principles to the direction of the robot machines which could then be produced. The first is clearly a practical project for engineer while the latter is a question of computing and software.

At present, real world robots are confined to highly specific tasks, perhaps a monotonous and fairly simple industrial application in conditions unsuitable for human workers, or a experimental research programme aimed at duplicating a particular human skill, like bipedal walking, in a machine. One ultimate goal of robotic research, however, will probably be the production of genuinely human-like androids capable of many of the skills and activities of ordinary people. Such an objective is necessarily placed in the distant future because of the complexity of the problems involved and the current limitations of computational power.

A more modest goal to be achieved before then will be the development of complex and relatively autonomous robots which, although lacking most of the abilities of a real person, will nevertheless be capable of purposive action of their own, directed towards a task set by human supervisors. Such robotic machines would be of immense use in undersea exploration and certainly in future space programs. Robot colonies on other planets may well be more feasible initially than human settlement.

No one research programme is currently directed towards the development of robots of this degree of complexity, although parts of the solution are being achieved independently of one another. For example, much research has been conducted into the creation of useful robotic vision and intelligent image analysis. This would permit a future robot to internalise a model of its environment in a analogous way to animal and human sight, although such demands are necessarily made upon computational power that far more complex computer architectures will be needed for a complete solution.

Searle, John

Professor of Philosophy at the University of California, Berkeley, Searle has become interested in many of the issues raised by A.I. He is critical of the position adopted by the proponents of strong A.I., those computer scientists who have claimed that genuine thought will eventually be achieved by appropriate machines and software. In attacking this belief, Searle has devised his Chinese room scenario. Here, a non-Chinese speaker

could be locked in a room with baskets containing Chinese symbols, together with a rule book explaining which of them should be passed back outside the room in response to other symbols handed in. It is assumed that the set of rules is sufficiently complex to permit an apparently coherent conversation to take place with the people outside the room. They would then make the plausible assumption that a native Chinese speaker was inside. Of course, they would be mistaken in this belief. Searle uses this simple thought-experiment to show that no computer program could ever create real understanding of language. He argues that although a computer program could conceivably contain all the necessary rules for communicating in a given language, true linguistic ability requires far more than syntax. Instead, a comprehension of language at a semantic level is required and it is hard to see how this could ever be acquired by a computer program. The relevance of this counter argument to the validity of the Turing test is obvious.

Shakey

'Shakey' was the name given to an early robot project at Stanford. It was a small robot vehicle controlled via a radio link to a PDP-10 computer and could explore and interact with its environment. Research was conducted into practical issues in vision, navigation and problem-solving. Some of the experience gained in the project was later applied to the design of NASA's successful Viking spacecraft which landed on Mars and photographed the surface.

SHRDLU

This was the natural language program written by Terry Winograd. It could understand perfectly ordinary English provided that all dialogue was restricted to the very simplified Blocks World. This consisted only of solid geometric shapes. The name of the program was chosen to be deliberately meaningless. It is not an acronym, but instead comes from the group of intentionally random letters once inserted by typesetters into a manuscript to indicate the location of a mistake.

Strong A.I.

This is John Searle's slightly derisory term for those A.I. theorists who regard the human mind as being equivalent to software running on a particularly powerful, biological computer.

Turing, Alan

An influential figure in the development of the first electronic computers was the British mathematician, Alan Turing (1912-54). A Fellow in Mathematics at Kings College, Cambridge, he published a paper in 1936, 'On Computable Numbers', in which he outlined an imaginary Turing machine. This was essentially a mathematical description of the fundamental principles subsequently employed in the design and programming of real computers. Turing was an essential member of the research team which created the code breaking Colossus computer during the Second World War. Later he worked on the development of computers at the National Physical Laboratory and at Manchester University, remaining there until his suicide. He was fascinated by the possibility of Artificial Intelligence and conducted theoretical research into A.I. long before practical programming was possible, prefiguring much of the work of the

decade following the war. Although his ideas now appear to have led in an unprofitable direction for A.I., his concept of the Turing test as a definition of intelligence is still often quoted.

Turing Institute

This is an organisation established by Donald Michie and others at Glasgow in 1983 and intended to encourage research in A.I. with a strong industrial orientation. The choice of name is a tribute to the importance of Alan Turing in the history of computer science and artificial intelligence.

Turing Machine

The initial conception of the Turing machine arose from Alan Turing's interest in a problem first posed in 1900 by the mathematician David Hilbert. He had challenged the International Congress of Mathematicians with a list of unsolved problems. The 23rd of these was to establish a method for establishing the truth or falsity of any statement in predicate calculus. The resolution of this problem had to await a later generation of mathematicians when, in 1936, Alan Turing published his response in a paper 'On Computable Numbers'. Here he demonstrated that Hilbert's problem could not, in fact, be solved. However, during the course of his argument, Turing needed to develop a formal model of the process of computation and this became the logical concept of the Turing machine. Since then, in addition to its role as a conceptual model in mathematical logic, the Turing machine has acquired an importance in the theoretical basis of computer science.

A Turing machine can be regarded quite simply in physical terms. Here it consists of an infinite tape which can move to the left or the right beneath a read/write head which can place a 1 or a 0 at each location on the tape, overwriting if necessary the symbol that is already there. The machine operates in a series of discrete actions. Each action is determined by two conditions, the current symbol on the tape and the prior internal state of the machine. The action itself consists of three separate instructions, the new symbol to leave on the tape, the modified internal state for the machine and whether the tape is to move one place to the left or the right. The entire operation of the machine can be defined as a table of these three-part instructions written, like a computer program, for some specific overall task. Despite the seeming emptiness of this formalism, the Turing machine can be programmed to complete complicated calculations, achieved by a combination of simpler Turing machines as building blocks.

Turing Test

Alan Turing hoped to create a yardstick for the definition of intelligence with his description of a possible method for deciding whether a particular combination of computer and programming could be assumed to be intelligent. In this he made the suggestion that if it is impossible to distinguish between a conversation with a real, though concealed, human being and a similarly hidden computer, then the machine can be regarded as genuinely thinking. Despite the simplicity and apparent plausibility of the test, it has been criticised in depth by philosophers like John Searle and Hubert Dreyfus.

Waltz Algorithm

This is used in computer vision and helps to label the vertices of a wireframe representation of an object in a consistent way. Practical constraints imposed on the vertices are exploited immediately to reduce the number of different possible ways of associating labels with them.

Weak A.I.

The philosopher John Searle describes two current schools of thought in A.I. research, which he terms 'strong' and 'weak' A.I., the latter making more modest claims for the potential of computers and not simply equating the human mind with very exotic software.

Winograd, Terry

A leading figure in A.I. research, Terry Winograd developed at M.I.T. the natural language program SHRDLU, an account of which appears in *'Understanding Natural Language'* (Edinburgh University Press, 1976). He is now continuing research as Professor of Computer Science and Linguistics at Stanford University.

Appendix B

REFERENCES

There is now a very wide range of literature dealing with various topics mentioned in this book. All of the following should prove useful in supplementing the material here and the titles been grouped to reflect the order in which concepts have been introduced. Most of the books are still in print and those that are not, will be available through libraries. In order to help trace books via booksellers, the ISBN number has been quoted. Where there has been a choice of binding, this refers always to the paperback edition.

The following books all introduce programming in LISP.

LISP: The Language of Artificial Intelligence by A.A. Berk, Collins, 1985, ISBN 0-00-383130-2.
This is an extremely readable book and begins with a lengthy section explaining the concept of the list structure in detail.

A Beginner's Guide to LISP by Tony Hasemer, Addison-Wesley, 1984, ISBN 0-201-14634-7.
Slightly more difficult than the above, Hasemer's book is intended for the beginner who is learning LISP programming on a personal computer. It is not aimed at any specific LISP dialect and advice is given on how to define any LISP functions assumed in the text, but not available with the reader's language. A noughts and crosses game is developed as a major programming exercise and a production system is also generated.

LISP-The Language of Artificial Intelligence by Frederick Holtz, Tab Books, 1985, ISBN 0-8306-0320-4.
Holtz's book, like this one, is about programming in Golden Common LISP and is extremely useful.

An Introduction to LISP by A. Narayanan and N. Sharkey, Ellis Horwood, 1985, ISBN 0 85312 735 2.
An extremely easy book to read, this makes no prior assumptions about the reader's programming skills and proceeds at a gentle pace. Fundamental ideas are explained carefully and illustrated with worked examples and exercises. It does not use any particular LISP dialect.

LISP by Patrick Winston and Berthold Horn, Addison Wesley, 1984, ISBN 0 201 08372 8.
This is an important standard work on LISP programming and is essential reading.

Further information on some of the applications developed in this book will be found in the following titles.

Data Structures and Algorithms by Alfred Aho, John Hopcroft and Jeffrey Ullman, Addison-Wesley, 1985 ,ISBN 0-201-00023-7.
A thorough, mathematical description of various data structures is given in this text with programming examples illustrated in Pascal-like 'pseudo-language'.

Basic Queueing Theory by Brian D.Bunday, Arnold, 1986, ISBN 0-7131-3570-0.
A detailed introductory course is given in queueing theory by this book, together with program examples in BASIC suitable for conversion into LISP.

Basic Statistical Programming by D.Cooke, A.H.Craven and G.M.Clarke, Arnold, 1984, ISBN 0 7131 3441 0.
This book is extremely useful in showing how a personal computer can be applied to elementary statistical analysis of data. Many helpful algorithms are explained and, although all the programming examples are in BASIC, it would not too difficult to recode them as LISP functions.

'Data Structures and Algorithms' by Niklaus Wirth, *Scientific American*, September, 1984.
This article is an interesting description of various data structures and their application.

Build your own Expert System by Chris Naylor, Sigma Press, 1983, ISBN 0-905104-41-2.
An entertaining explanation of expert systems is given with practical BASIC listings worth converting to LISP.

Expert Systems - A Practical Introduction by Peter S. Sell, Macmillan, 1985, ISBN 0-333-37264-6.
A good illustration of production systems is provided here.

APPENDIX C

SOLUTIONS

Exercise 1

1. The expression was:

(BACH ((BARTOK) ((BEETHOVEN) BOULEZ) (BRAHMS)) BRITTEN)

This is a valid list containing three elements. These are:

```
BACH
((BARTOK)  ((BEETHOVEN)  BOULEZ)  (BRAHMS))
BRITTEN
```

The first and last of these are atoms, but the second is a sublist of some complexity. It can be analysed itself into three elements, which are:

```
(BARTOK)
((BEETHOVEN)  BOULEZ)
(BRAHMS)
```

The first and third are single element lists. The second is another sublist with two elements:

```
(BEETHOVEN)
BOULEZ
```

The first element is a single element list, while the second is an atom.

It is important to be able to analyse an expression in LISP in this fashion in order to understand precisely what is happening inside it and how the expression will evaluate in different circumstances. If, on the other hand, the expression is simply being checked very quickly in order to see whether the total numbers of opening and closing brackets balance, the faster method explained earlier can be used. In this, you work from the beginning of the list to the end and keep an overall score which shows the excess of opening brackets over closing. You could, of course, operate with the scoring working the other way round and count by how many closing brackets exceed opening. The important point is that the final tally, when the end of the expression is reached, should be zero. If it is not, then a valid list with balanced brackets is not present. This method of testing the validity of a potential symbolic expression is illustrated below. Of course, you would perform the mental arithmetic in your head, but, if you did

decide to jot down how you were working, it could be recorded schematically as shown. The total number of excess opening brackets is represented here by the value of the variable S.

```
                (BACH ((BARTOK) ((BEETHOVEN) BOULEZ) (BRAHMS)) BRITTEN)
  S=1
                  ((BARTOK) ((BEETHOVEN) BOULEZ) (BRAHMS)) BRITTEN)
    S=2
                  (BARTOK) ((BEETHOVEN) BOULEZ) (BRAHMS)) BRITTEN)
     S=3
                      ) ((BEETHOVEN) BOULEZ) (BRAHMS)) BRITTEN)
            S=2
                      ((BEETHOVEN) BOULEZ) (BRAHMS)) BRITTEN)
                S=3
                      (BEETHOVEN) BOULEZ) (BRAHMS)) BRITTEN)
                   S=4
                            ) BOULEZ) (BRAHMS)) BRITTEN)
                              S=3
                               ) (BRAHMS)) BRITTEN)
                                        S=2
                               (BRAHMS)) BRITTEN)
                                          S=3
                                    )) BRITTEN)
                                                  S=2
                                    ) BRITTEN)
                                                   S=1
                                        )
                                                                  S=0
```

The overall value of S for the whole expression is zero, as it should be for a valid expression with balanced brackets. Either of these two methods can now be applied to the other expressions in the exercise.

2. The expression was:

```
(GERSHWIN   (GLASS   GLUCK   GOTTSCHALK   GRAINGER   (GRIEG))
GRUBER)
```

This again is a valid list. It has these elements:

```
GERSHWIN
(GLASS GLUCK GOTTSCHALK GRAINGER (GRIEG))
GRUBER
```

The first and last of these are atoms but the second element is a sublist with five elements:

```
GLASS
GLUCK
GOTTSCHALK
GRAINGER
(GRIEG)
```

All these elements are atoms, except for the last which is a single element list.

3. The expression was:

```
((HANDEL     ((HARRISON-BIRTWISTLE)     ((HAYDN)     HENZE)))
HINDEMITH)
```

This also is a valid list, containing two elements:

```
(HANDEL ((HARRISON-BIRTWISTLE) ((HAYDN) HENZE)))
HINDEMITH
```

The second element is an atom. The first is a sublist which again has two elements:

```
HANDEL
((HARRISON-BIRTWISTLE) ((HAYDN) HENZE))
```

The second of these elements is a further sublist which can be analysed into:

```
(HARRISON-BIRTWISTLE)
((HAYDN) HENZE)
```

The first element here is a single element list and the second a two element list consisting of another single element list and an atom:

```
(HAYDN)
HENZE
```

This example indicates the type of hierarchy possible in LISP coding, which, remember, is one of the principal reasons for the concise power of the language.

4. The expression was:

```
(((MAHLER) (MAXWELL-DAVIES) (MENDELSSOHN))) MOZART)
```

This is not a valid list, as can be shown by the method of counting brackets:

```
(((MAHLER) (MAXWELL-DAVIES) (MENDELSSOHN))) MOZART)
  S=1
((MAHLER) (MAXWELL-DAVIES) (MENDELSSOHN))) MOZART)
    S=2
(MAHLER) (MAXWELL-DAVIES) (MENDELSSOHN))) MOZART)
      S=3
) (MAXWELL-DAVIES) (MENDELSSOHN))) MOZART)
        S=2
(MAXWELL-DAVIES) (MENDELSSOHN))) MOZART)
          S=3
) (MENDELSSOHN))) MOZART)
                          S=2
(MENDELSSOHN))) MOZART)
                             S=3
))) MOZART)
                                          S=2
)) MOZART)
                                            S=1
) MOZART)
                                              S=0
)
                                                  S=-1
```

The final value of `-1` indicates that there is one more closing bracket than opening and that therefore the expression is not balanced.

5. The expression was:

```
((WAGNER) (WALTON) (WEBER (WEBERN WEILL WOLF))
```

This is an unbalanced expression. The overall list has the elements:

```
(WAGNER)
(WALTON)
(WEBER (WEBERN WEILL WOLF)
```

The first two are legitimate single element lists, but the final element is an unbalanced list, containing two opening brackets, but only one closing. Therefore the initial expression is not valid.

6. In this example the expressions was:

```
(WATER SUGAR CITRIC-ACID ANTI-OXIDANT E104 E110 E150)
```

The total number of elements is seven because the hyphens linking `CITRIC-ACID` and `ANTI-OXIDANT` create single atoms in each case.

7. Similarly, because of the hyphens, there are only three elements in the list:

```
(EINSTEIN-ON-THE-BEACH SATYAGRAHA AKHNATEN)
```

(These, incidentally, are the names of three operas by the American composer, Philip Glass.)

8. The calculation which needs to be performed is:

```
(23 * 5) + (13 - 6)
```

Three stages need to be completed. The product of 23 and 5 has to be found. After this the difference between 13 and 6 is calculated. Then the two answers must be added together. All of this has to be done in the appropriate order and using reverse notation. Of course, the subtraction could be carried out before the multiplication, but below the same order as indicated in the complete calculation will be adopted.

The LISP expression which multiplies 23 and 5 is:

```
(* 23 5)
```

Reverse notation is used with the function appearing at the head of the list, followed by the two numerical arguments.

The subtraction is then performed by the expression:

```
(- 13 6)
```

Reverse notation is used for a third time in assembling the two sublists into the expression which carries out the overall calculation:

```
(+ (* 23 5) (- 13 6))
```

The expression will automatically have the correct distribution of opening and closing brackets.

```
(+ (* 23 5) (- 13 6))
                                    S=1

(* 23 5) (- 13 6))
                                      S=2

) (- 13 6))
                                         S=1
```

```
(- 13 6))
```

S=2

```
))
```

S=1

```
)
```

S=0

Entering the expression into LISP will lead to evaluation of the correct answer:

```
* (+ (* 23 5) (- 13 6))
122
```

9. The calculation is:

```
(4 + 6) * (25 / 5)
```

The addition is performed by:

```
(+ 4 6)
```

The division is calculated by the expression:

```
(/ 25 5)
```

Combining the two separate lists as the arguments for the multiplication function gives the required expression:

```
(* (+ 4 6) (/ 25 5))
```

This will be evaluated by LISP to yield the answer `50`.

10. The calculation is:

```
(12 + 2) * (7 / (8 - 3))
```

There are three separate calculations which must be carried out before the final multiplication. Their order is significant. First, the subtraction must be evaluated as:

```
(- 8 3)
```

The answer to this can then be inserted into the further expression involving the division:

```
(/ 7 (- 8 3))
```

The addition takes place separately as:

```
(+ 12 2)
```

When all three individual calculations have been coded into LISP the final expression can be written:

```
(* (+ 12 2) (/ 7 (- 8 3)))
```

This evaluates to 33.6.

11. The obvious name to choose for a function which is going to calculate the cube of a number is CUBE. It is not necessary, of course, to make this choice. LISP allows you considerable freedom and so you might want your function to be known as THIRD-POWER, or some other name which was convenient.

The function will need to accept a number typed in by the user. A suitable variable to use here is N, although again it could be X or another variable according to preference. This variable will need to be on the parameter list and so the general form of the function will be:

```
(DEFUN CUBE (N)
 (Body of definition))
```

How the body of the definition itself is written is again a matter of individual choice. If the function SQUARE has already been included in the LISP environment, this can be incorporated into CUBE. Multiplying a number by its square gives the cube of the number. Adopting this approach gives the definition:

```
(DEFUN CUBE (N)
 (* N (SQUARE N)))
```

Alternatively, the number can just be multiplied by itself the appropriate number of times:

```
(DEFUN CUBE (N)
 (* N (* N N)))
```

Note that in both definitions the brackets are balanced.

Exercise 2

1. Here a function called RAFFLE has to be written which will respond to the user's ticket number by printing either 'You have won' or 'Bad luck' on the screen.

This function will then be evaluated simply by typing:

```
* (RAFFLE)
```

RAFFLE is defined using the LISP primitive DEFUN. Recall how this function operates. The first element in the expression employing it is the function name itself, DEFUN. After this, DEFUN's argument consists of three parts, the name of the new function being defined, the list of arguments that this requires and finally the symbolic-expressions which evaluate to give the intended result.

The first two items in the definition will therefore be DEFUN followed by RAFFLE. The parameter list for RAFFLE must be NIL because the function, as illustrated above, does not require any argument to be added before the closing bracket. The outline form of the function must be:

```
(DEFUN RAFFLE NIL
 (Body of function))
```

As the response of the function to the user's raffle ticket number will be to print either 'You have won' or 'Bad luck', a simple IF condition is sufficient. First, RAFFLE must request the number by using (PRINT '(TYPE YOUR TICKET NUMBER)). The user's input is then ascertained by READ as part of the condition in the IF expression which follows. The winning number is coded directly into the condition as well. Here is the complete definition:

```
(DEFUN RAFFLE NIL
 (PRINT '(TYPE YOUR TICKET NUMBER))
 (IF (= (READ) 4507)
  (PRINT '(YOU HAVE WON))
   (PRINT '(BAD LUCK))))
```

As always, you should check that the brackets lead to a balanced expression for the new function definition. Brackets have an inevitable tendency to cluster in an alarming fashion at the end of a definition and so it is worthwhile to assure yourself that they are all there for good reason. In this particular example, there are four closing brackets at the very end. Their role can be re-established quickly, reading them from left to right. The first is then the closing bracket for the list (BAD LUCK). The second balances the bracket before the PRINT. The third marks the end of the conditional expression and finally the last bracket matches the initial opening bracket before the DEFUN.

The new function will now operate like this:

```
* (RAFFLE)
(TYPE YOUR TICKET NUMBER) 1248
(BAD LUCK)
(BAD LUCK)
```

```
* (RAFFLE)
(TYPE YOUR TICKET NUMBER) 4507
(YOU HAVE WON)
(YOU HAVE WON)
```

2. The loop is:

```
(DO ((X 7 (+ 4 X)))
    ((> X 30))
    (PRINT X))
```

The values of X displayed will begin at 7 and then increment by 4, until 27 is reached. The next value would be 31, but this causes the loop to terminate. These will be the numbers shown: 7, 11, 15, 19, 23, 27

3. The first ten positive numbers which the required function must generate are:

```
12, 48, 108, 192, 300, 432, 588, 768, 972, 1200
```

It is interesting to see how this result is produced by a two variable DO loop as suggested and is good practice in the sort of algorithm that is often useful when writing functions which are going to involve arithmetical patterns. The function, NUMBERS, defined as shown uses a DO loop in which the variables X and Y increment by 3 and 4 to provide the necessary pattern:

```
(DEFUN NUMBERS NIL
 (DO ((X 3 (+ X 3))
      (Y 4 (+ Y 4)))
         ((> X 30))
         (PRINT (* X Y))))
```

4. PARITYP has to be a predicate function, returning T for even numbers and NIL for odd. Two clues were given, the use of FACTORP and the fact that 2 is always a factor of even numbers.

The use of FACTORP means that both of the following definitions have to be entered into the LISP environment before PARITYP can be written:

```
(DEFUN INT (N)
 (SETF NA N)
 (DO ((X 1 (1+ X)))
     ((< N 0))
     (SETF N (1- N)))
 (- NA N 1))

(DEFUN FACTORP (A B)
 (= (/ B A) (INT (/ B A))))
```

The algorithm is then quite simple. If the number, N, being tested is even, it will have 2 as a factor. This means that (FACTORP 2 N) will return T. If the number is odd, 2 will not be a factor of it. (FACTORP 2 N) will therefore evaluate as NIL. This means that the function definition for PARITYP will be:

```
(DEFUN PARITYP (N)
 (FACTORP 2 N))
```

Testing the function gives:

```
* (PARITYP 23)
NIL

* (PARITYP 18)
T
```

Exercise 3

1. The expression to be evaluated was:

```
(APPEND '(A B) '(C D) '(E F))
```

This evaluates to:

```
(A B C D E F)
```

2. The expression to be evaluated was:

```
(LIST '(A B) '(C D) '(E F))
```

This evaluates to:

```
((A B) (C D) (E F))
```

The distinction between APPEND and LIST is clear in these examples.

3. The expression to be evaluated was:

```
(FIRST '(ALAN BOB CHRIS DAVID))
```

This evaluates to:

```
ALAN
```

4. The expression to be evaluated was:

```
(SECOND '(A (B C) D E))
```

This evaluates to:

```
(B C)
```

5. The expression to be evaluated was:

```
(NTH 3 (REVERSE '(A B C D E F G H)))
```

Here, `(REVERSE '(A B C D E F G H))` is evaluated first to give the new list `(H G F E D C B A)`. `NTH` then gives element number three in this list, beginning from `0`. The expression therefore evaluates to `E`.

6. Rather than have a single function which stores details of phone calls and their duration, it is simpler to have three separate functions, `START`, `RECORD`, and `FIND`. These functions need only employ one array, `DETAILS`, to store the name of the person phoned and the length of the call. This can be done by having adjacent array locations holding alternately the name of the person and the number of minutes. In order to simplify the description of the functions, the assumption is made here that there are only five calls recorded altogether. This means that the array needs only ten locations and so can be set up by `(MAKE-ARRAY 10)`. Of course, this array would probably be larger in practice. The total number of calls to be made is recorded as the variable `P`. The initial value of `P` before any calls have been recorded must be `-1`. This is because `P` will be incremented by 1 for each call and so, if the array location 0 is to be used, `P` must begin as the negative value `-1`. After it is incremented it will then identify the first array location, `0`. Both `P` and `DETAILS` can be set up independently of the two functions which record and retrieve information respectively by the function `START`. This is coded as follows:

```
(DEFUN START NIL
 (SETF P -1)
 (SETF DETAILS (MAKE-ARRAY 10)))
```

The function which initially stores the details of a particular call is `RECORD`. In coding this, the relationship between the values of `P` and the array locations of the name and number for each call needs to be known. For the first, three calls this can be seen to be like this:

P=0		P=1		P=2	
NAME	LENGTH	NAME	LENGTH	NAME	LENGTH
0	1	2	3	4	5

The array location for the name is just twice the value of `P` and the array location of the number is one more than this. The function `RECORD` can therefore be coded to incorporate this simple arithmetical relationship:

```
(DEFUN RECORD NIL
 (SETF P (1+ P))
 (PRINT '(TYPE NAME))
 (SETF (AREF DETAILS (* 2 P)) (READ))
 (PRINT '(TYPE NUMBER OF MINUTES))
 (SETF (AREF DETAILS (1+ (* 2 P))) (READ)))
```

PRINT is used to request the information, which is then obtained by READ. The arithmetical expressions embedded in the two calls to the array DETAILS calculate the appropriate location each time. The function will operate as illustrated:

```
* (RECORD)
(TYPE NAME) MARY
(TYPE NUMBER OF MINUTES) 3
```

Now the last of the three functions, FIND, needs to be coded.

```
(DEFUN FIND NIL
 (PRINT '(TYPE NAME))
 (SETF NAME (READ))
 (DOTIMES (X 5 T)
          (IF (EQUAL (AREF DETAILS (* X 2))
           NAME) (PRINT (AREF DETAILS
            (1+ (* X 2)))))))
```

This function is able to extract the required information from the array DETAILS. First, the name of the person to whom the call was made must be typed in. This is assigned to NAME. The DOTIMES loop then checks through the array to see if the person is matched with any of the stored names. EQUAL does this. If a name does match, the next array location is printed out.

The nest of brackets at the end of the function may need some clarification. Starting from the left, the first bracket completes the doubling of the value of X. The next corresponds to the incrementing by 1 and the third marks the end of the array reference. The fourth corresponds with the PRINT and the fifth ends the IF conditional. The next bracket ends the DOTIMES loop and the last marks the end of the function definition.

Here is FIND in operation:

```
* (FIND)
(TYPE NAME) MARY
3
T
```

And this is the result if a name is entered which is not in the array:

```
* (FIND)
(TYPE NAME) BERTIE
T
```

7. SIDE will take as its argument the lengths of two sides of a right-angled triangle, one of which must be the hypotenuse. It will then calculate the length of the third side. Either of the sides given may be the hypotenuse. Using the familiar '3-4-5' example of a right-angled triangle, this means that SIDE will operate like this:

```
* (SIDE 3 5)
4.0D0

* (SIDE 5 4)
3.0D0
```

It can be seen that in the first example the hypotenuse was the second number following SIDE. In the other example, it came first. A simple conditional can be used to decide which of the two sides entered is the hypotenuse. The order of the variables in the Pythagorean calculation following will then be adjusted accordingly. SIDE needs to use the function defined earlier, SQROOT:

```
(DEFUN SIDE (A B)
 (IF (> A B) (SQROOT (- (* A A) (* B B)))
  (SQROOT (- (* B B) (* A A)))))
```

Exercise 4

1. The expression to be evaluated was:

```
(NULL (< 9 4))
```

Since 9 is greater than 4, the sublist evaluates as NIL and so T is returned by NULL in the overall expression.

2. The expression to be evaluated was:

```
(NULL (= (* 3 4) (1+ 11)))
```

As 3 multiplied by 4 is the same as 11 plus 1, the argument for NULL evaluates as T and so NIL is returned by the whole expression.

3. The expression to be evaluated was:

```
(NULL (EQUAL (REST '(A B C D)) '(B C D)))
```

(REST '(A B C D)) is (B C D) and so the two arguments for EQUAL have the same structure. NULL therefore has T as its argument and returns NIL.

4. TERPRI and FORMAT T will be used like this in the function definition:

```
(DEFUN BARD NIL
 (TERPRI)
 (FORMAT T
  "Now is the winter of our discontent")
 (TERPRI)
 (FORMAT T
  "Made glorious summer by this son of York")
 (TERPRI)
 (FORMAT T
  "And all the clouds that loured upon our house")
 (TERPRI)
 (FORMAT T
  "In the deep bosom of the ocean buried"))
```

The display will be as follows.

```
* (BARD)
Now is the winter of our discontent
Made glorious summer by this son of York
And all the clouds that loured upon our house
In the deep bosom of the ocean buried
```

5. The function SCROUNGE will be coded like this:

```
(DEFUN SCROUNGE NIL
 (TERPRI)
 (FORMAT T "Will you lend us five quid till
  Thursday?")
 (TERPRI)
 (DO ((X 1 (1+ X)))
     ((Y-OR-N-P))
     (TERPRI)
     (FORMAT T "Will you lend us five quid till
      Thursday?")
     (TERPRI))
 (TERPRI)
 (FORMAT T "Cheers"))
```

FORMAT T is used to place the initial demand on the screen. It is then repeated indefinitely by the loop until the exit condition evaluates as T, leading to this display:

```
* (SCROUNGE)
Will you lend us five quid till Thursday?
(Y or N) No
```

```
Will you lend us five quid till Thursday?
(Y or N) No
Will you lend us five quid till Thursday?
(Y or N) No
Will you lend us five quid till Thursday?
(Y or N) No
Will you lend us five quid till Thursday?
(Y or N) Yes
Cheers
```

Exercise 5

1. The expression to be evaluated was:

```
(AND (= 5 5) (> 7 2) (< 3 9))
```

The first argument for the AND is `(= 5 5)`, which evaluates as `T`. In the same way, the second argument, `(> 7 2)`, and the third argument, `(< 3 9)`, are both true as well. As all of its arguments have the value `T`, the overall expression returns T.

2. The expression to be evaluated was:

```
(AND (NULL (= (* 2 7) (/ 26 2))) (> 5 4) T)
```

The first argument of the AND is this expression:

```
(NULL (= (* 2 7) (/ 26 2)))
```

The most deeply embedded sublists evaluate as `14` and `13`. As these are not equal, the argument for `NULL` has the value `NIL` and so `NULL` returns `T` for the first argument of AND. The second argument for AND, `(> 5 4)`, returns `T` and the last argument is already T. The overall expression therefore returns `T`.

3. The expression to be evaluated was:

```
(AND (NULL (MEMBER 'A '(B C D))) (EQUAL '(C D)
 (MEMBER 'C '(B C D))) (= 5 (1- 6)))
```

The first argument for the AND is:

```
(NULL (MEMBER 'A '(B C D))).
```

As A is not a member of the list, the `NULL` reverses the value of `NIL` to `T`.

The second argument of the AND is:

```
(EQUAL '(C D) (MEMBER 'C '(B C D)))
```

(MEMBER 'C '(B C D)) evaluates as (C D), so this argument evaluates as T. The third argument is (= 5 (1- 6)). This again is T, so the whole AND expression returns T as each of its arguments are T.

4. The function could be written as:

```
(DEFUN LEADER NIL
 (TERPRI)
 (FORMAT T "Type name of politician")
 (TERPRI)
 (SETF LEADER (READ))
 (COND ((EQUAL LEADER 'MAGGIE)
        (FORMAT T "She leads the Conservatives"))
       ((EQUAL LEADER 'NEIL)
        (FORMAT T "He leads the Labour Party"))
       (T (FORMAT T "Oh the SLD leader"))))
```

5. An appropriate function is:

```
(DEFUN MATHS (N)
 (COND ((> N 7) (* N 5))
       ((= N 3) 1)
       (T (+ N 23))))
```

6. The programme could be placed on the property list by this function:

```
(DEFUN RECORD-PROG NIL
 (TERPRI)
 (FORMAT T "Type day ")
 (SETF DAY (READ))
 (TERPRI)
 (FORMAT T "Type programme ")
 (SETF TITLE (READ))
 (PUTPROP 'CHOICE (CONS TITLE (GET 'CHOICE DAY))
  DAY))
```

The selection of programmes for any day of the week could be examined by this function:

```
(DEFUN RECALL-PROG NIL
 (TERPRI)
 (FORMAT T "Type day ")
 (SETF DAY (READ))
 (TERPRI)
 (FORMAT T "Programme selection is ")
 (GET 'CHOICE DAY))
```

Exercise 6

1. The expression to be evaluated was:

```
(ZEROP (- (* 3 4) (* 6 2)))
```

The argument for ZEROP evaluates first as (- 12 12), and then zero. Hence T is returned.

2. The expression to be evaluated was:

```
(ZEROP (* (+ 4 7) (- 2 9)))
```

Here, the argument becomes (* 11 -7) which is not zero. Therefore NIL is returned.

3. The expression to be evaluated was:

```
(NOT (ZEROP (- 20 (* 4 5))))
```

The argument for ZEROP becomes (- 20 20), so T is returned. The overall expression then evaluates as NIL.

Exercise 7

1. Here, MAPCAR is used with a lambda expression in a function, SQLIST, which squares the elements of a numerical list. The list has the identifier NUMLIST.

```
(DEFUN SQLIST (NUMLIST)
 (MAPCAR #'(LAMBDA (N) (* N N)) NUMLIST))
```

Here is an illustration of the function in use:

```
* (SQLIST '(1 2 3 4 5 6)
(1 4 9 25 36)
```

2. Here, a function called CORE is written which uses BUTLAST and NTHCDR to return the middle of a list without the first and last pairs of elements.

The list is represented by L. Then (BUTLAST L 2) will return the list without its last two elements. The first two elements are removed as well by using (NTHCDR 2 (BUTLAST L 2)). The whole function is therefore defined as:

```
(DEFUN CORE (L)
 (NTHCDR 2 (BUTLAST L 2)))
```

It then operates like this:

```
* (CORE '(ONIONS CARROTS CELERY CABBAGE TOMATOES))
(CELERY)
```

INDEX

a-list, 110
A.I., (see Artificial Intelligence)
ABS, 65, 69
accountability, 205, 234
ACONS, 110, 132
ADD-CHILD, 137
ADD-NODE, 124
AKIN, 195, 196
algorithm, sorting, 61
algorithms, 23
ALIKE, 195-197, 207
alter, 183
analysis, input-output, 86
AND, 120, 132
APPEND, 44, 51, 69
ARC, 124, 125, 126
arcs, 111, 218
AREF, 57, 69
arguments, 9, 10, 17, 19, 21
Aristotle, 117, 122, 123, 128, 131, 219
arithmetic operations, 9, 15
array, 57, 102
array, fill, 58
array, make, 69
array, mean, 59
array, median, 62
array, show, 58, 59, 61
array, sort, 60, 63
arrays, multi-dimensional, 57
arrival mechanism, 145
artificial intelligence, 5, 119, 172, 216, 222
Asimov, Isaac, 260
ASK, 233, 234, 238
assignment, 10
ASSOC, 110, 132
assocation, 126, 132
association list, 102, 109, 111
atoms, 8, 11, 14, 21
augmented transition network, 256
automatic mathematician, 250
average, 9, 12, 13, 17, 17

backward chaining, 221
bacteria, 176, 177, 179, 188, 189, 190
BASIC, 10, 24, 42, 79, 176
Berkeley, 257
binary trees, 111, 115, 135
binding, 10, 21
black boxes, 117, 224
blank, 163
blind search, 139
blocks world, 247, 261
Boden, Margaret, 247
Boltzmann architecture, 248
Boolean value, 26, 100, 129, 136
bracket, 8, 12, 13, 18, 20
branch, 114
branching, 23, 28, 40
breadth-first search, 141, 142
bubble sort, 60
BUILD-QUEUE, 149, 151, 154, 165, 165
BUILD-TREE, 136
BUTLAST, 182, 193

C-To-F, 18
C, 79
calculations, mixed, 16
Camus, 256
Capek, Karel, 248
CAR, 46
Carnegie-Mellon University, 118
CDR, 49
cells, 181
Celsius, 18
central-state materialism, 257
centred trend, 72, 75
Charniak, Eugene, 254
check-out, 145, 164, 165
Chinese room, 260
Chomsky, Naom, 253
Church, Alonzo, 252
Clarke, Arthur C., 253
clock, 155
code-shape, 206, 207
coerce, 173, 174
cofactor, 94
cognitive science, 248
column matrix, 99
column vector, 98
combinatorial explosion, 196
Common Lisp, 9, 11, 252
compensating causal loop, 172
composite numbers, 37
computational model, 171
computer vision, 6, 194, 196, 215, 263
COND, 119, 132
conditional expression, 28, 29, 30, 36
conditions, 146, 148, 154
conference, Dartmouth, 247
congruence method, 156
connection machine, 254
CONS, 52, 69, 104
continuity, 203
control structure, 23, 26

CTRL, 11, 12
cybernetics, 253
cycle, 179, 187
cyclical variation, 70

Dartmouth Conference, 5, 119, 247
data, 73
data structures, 6, 102, 116, 132, 134
database, 105, 108, 109
default slot, 217
DEFUN, 17, 18, 21, 39, 45
DELETE, 49, 51, 130, 132
Dendral, 251
Dennett, Daniel, 258
depth-first search, 140-143
Descartes, Rene, 37, 253, 257
descendants, 126, 127
determinant, 91-94
determinant, stored, 95
deterministic, 145
DEV, random, 74
deviation, 68
deviation, standard, 67
dialects, 27
disclose, 126
discontinuity, 203
DISH, 179, 180, 189, 197
dispersion, 67
display, 81
display-cells, 179, 184, 189
DO loop, 29-32, 43, 45
DO TIMES, 24, 25, 39, 40, 48
Dreyfus, Hubert, 250, 262
dumb search, 139
dynamic equilibrium, 176

edges, 215, 250
elements, 8, 20, 21
Eliza, 173, 224, 250
empty list, 15
entry conditions, 218
ENTRY, 120
EQUAL, 43, 69
equality, 27
Erlang, 134
error message, 11, 13, 16
error trapping, 169
ERROR, 11
Euclid, 37
Eurisko, 249, 250
EVAL, 183, 184, 188, 193, 199
evaluation, 10, 10, 12, 13, 14, 20, 21
EVENP, 53, 63, 69
exit condition, 24, 45
EXIT, 20, 21
expert system, 216, 220, 222-225, 251-252
EXPLODE, 172, 175, 193

FACTORP, 34
Fahrenheit, 18, 42
FALSE, 26, 28, 41

feedback, 222, 226, 246
FIFO, 134
fifth generation, 247, 251, 259
fill array, 58
fill tree, 113
find-perfect, 40
First In, First Out, 134
first, 46, 47, 69
FLAT, 79, 100
flowchart, 23
Fodor, Jerry, 258
for-next loop, 24
Forbidden Planet, 118
FORMAT, 100, 106
FORMAT T, 72
FORTRAN, 79
forward chaining, 221
four-point moving average, 72
frames, 217, 246, 252
front end, 120
function, 9, 10, 12, 13, 16, 21
functionalist model, 258
functions, predicate, 28
functions, recursive, 168

general tree structure, 135
GET, 102, 104, 132
get-decoded-time, 169
Golden Common LISP, 10-13, 16, 19-21, 24, 35, 42, 47, 57, 79, 92
GREATERP, 27
GREATEST, 49, 50, 52, 197

HALO, 79
hands-on, 10
heading, 81, 82
Hegel, 257
heuristic information, 216
heuristic search, 248
HEX, 57
hierarchy (of frames), 217
Hilbert, David, 262
Hitch Hiker's Guide, 29
homunculus, 258
human consciousness, 256
human cortex, 194
human expert, 216, 221, 251
Hume, Cyril, 118
Hume, David, 257
Husserl, Edmund, 259
hypercube, 255
hypothetical syllogism, 117, 130

IBM-PC compatible, 10, 20, 42, 155
idiot savant, 130
IF, 27, 41, 120
ignorant, 126, 127
implode, 175, 193, 197, 207, 209
indent, 81
indicator, 102
induction, 224

inequality, 27
inference engine, 221, 246
input matrix, 91
input-output analysis, 6, 86
input-output matrix, 88
input, 19, 44, 45, 48, 55
inputs, 87
INT, 32
INTEGER, 33, 39, 165, 197
integer arithmetic, 16
INTEGERP, 27, 41
Intelligent Mobile Platform, 118
intelligent software, 205
intentionality, 257
interactive, 11
InterLisp, 252
internal clock, 155, 169
internal representation, 225
inverse matrix, 91, 94, 95, 96, 99
iterative structure, 29

join-queue, 148, 154
justification, 120

key, 110
knowledge engineering, 251
knowledge representation, 217, 248, 252
knowledge-base, 217, 222, 246, 251

lambda, 174, 198
lambda calculus, 252
lambda expressions, 174, 193
last, 184, 193
Le Lisp, 252
leaf nodes, 111
learning algorithm, 228
left, 114
Lenat, 249, 250
length, 47, 48, 69
LESSP, 27
life, 176
light-up, 197, 199, 200, 201
line, 80
lines, 202
LISP dialects, 9, 15
LISP environment, 14, 18, 197
LISP machine, 252
list empty, 15
list head, 14
LISt Processing, 6
list, 6, 9, 12-14, 20, 44, 69
listener, 11, 16, 17, 19
lists, 132
locate, 197, 204
logarithm, 160
Logic Theorist, 5
LOGO, 252
looping, 23, 40
LOOPINT, 26

machine intelligence, 259
machine learning, 6, 196, 221, 246, 253
MacLisp, 252
make-array, 57, 69
Mann, Thomas, 249
MAPCAR, 173, 182, 193, 198, 207
Mars, 241, 242, 261
matrices, multiplying, 97, 99
matrix, 94
matrix form, 89
matrix, column, 99
matrix, inverse, 99
McCarthy, John, 5, 119, 247, 252, 254
McDermott, Drew, 254
mean, 46 - 48
mean array, 59
media cybernetics, 79
median array, 62
median, 46, 49, 50, 50, 51, 53
Meiko computing surface, 255
MEMBER, 124, 132
Michie, Donald, 262
mid-square algorithm, 157
Minsky, Marvin, 254
MINUS, 15
mixed calculations, 16
MODE, 46, 53, 54, 56
modify, 236
Moon, 241, 242
Moses, Joel, 216
MS-DOS, 20, 79
multi-dimensional arrays, 57, 92
multiplying (matrices), 97, 99
MYCIN, 251

NASA, 194, 261
natural language, 172, 247
negative feedback, 172
neighbours, 179, 185, 186, 187
nesting, 32
networks, 116, 132, 252
Newell, Allen, 5
Newton-Raphson, 64, 66
NIL, 15, 27, 39, 41
nodes, 111, 218
NOT, 136, 169
NTH, 46, 47, 53, 69
NTHCDR, 193
nuclear winter, 171
null, 15, 83, 100, 100
number crunching, 69
numbers, composite, 37
numbers, perfect, 36

Occam, 255
opinion, 235, 238
Oracle, 224, 238, 242
order, 49, 50, 50, 52, 55, 197
output, 19, 20, 87

page mode, 81
Papert, Seymour, 252

parallel processing, 247, 254
parameter list, 17, 25, 26, 33, 39, 45
parent node, 112
parenthesis, 18
parsing, 255
Pascal, 10, 79
pattern matching, 215, 256
pattern recognition, 5, 194, 215

pause, 80, 81, 165, 197
Perceptron, 253
perfect numbers, 36
PERFECTP, 37, 39
philosophical issues, 256
pixels, 196, 197, 199, 201, 249
Plato, 253
PLUS, 9
predicate, 34, 41
predicate functions, 26, 27, 202
prefix notation, 9, 15
prime numbers, 34, 35
PRIMEP, 25, 34, 36
PRIMES-TO, 35, 36
primitives, 6, 9, 12, 17, 27
prin1, 58, 69
print, 20, 21, 29, 72
print inverse, 95
print-queue, 152
problem solving, 6
procedural attachment slot, 217
production rules, 217, 219, 246, 252
production systems, 248
program structure, 23
programming top-down, 48
prolog, 247, 259
property, 102, 103, 126, 132
property list, 102, 104, 126,136, 197
Prospector, 251
pseudo-random numbers, 156
Putnam, Hilary, 259
PUTPROP, 102, 104, 132, 136, 181
Pythagorean relationship, 207

QUART-DEV, 74-77
quarterly figure, 72
query, 120, 130
question, 122, 129, 130, 231, 232, 238
queue, 134, 140, 141, 143
queue data structure, 134, 169
queueing discipline, 145
queueing theory, 6
queuing model, 134
queuing system, 145
queuing theory, 134
quote sign, 14
quote, 21
QUOTIENT, 15

Raibert, Marc, 259
RAND, 158
RANDOM DEV, 74, 78

random number algorithms, 169
random variation, 70
random, 157
randomise, 160
read-eval-print loop, 13
READ-LINE, 230, 232, 246
READ, 19, 20, 21, 26, 43
recognise, 197, 206, 207, 209
recognition, 6
record, 197, 204-206
records, 132
recursion, 23, 167
recursive functions, 168
reference slot, 217
reinforcing, 172
remove, 49, 50, 52, 197
REMPROP, 109, 132
REPEAT-UNTIL, 24
REPEAT, 120
reply, 115
request, 122, 126-128
RESET, 147, 148, 154
REST, 49, 49, 50, 69
results, 218
retina, 196, 197, 210, 211
retrieve, 105, 107
reverse, 106, 107, 132
right, 114
Robby, 118
robotics, 247, 260
robots, 118, 222, 259
roles, 218
root node, 135
root, 111
Rosenblatt, 253
Russell, Steve, 5

Sagan, Carl, 171
Saint Augustine, 37
Satre, 256
scenes, 218
scientific notations, 64
screen display, 79
scripts, 217, 246, 252
search techniques, 139
search tree, 115
search, blind, 139
search, breadth-first, 141
search, depth-first, 140
search, dumb, 139
Searle, John, 257, 260-263
seasonal variation, 70
SEASONAL-DEV, 77
SECOND, 46, 47, 69
sectors, 87
seed value, 161
seed, 156, 179, 183
semantic networks, 217, 218, 246
SERVER-STATUS, 162, 165
service mechanism, 145
service time, 150, 150

SERVICE-TIME, 151, 154
servicing event, 162, 163
SET, 179, 193, 198
SETF, 10, 14, 19, 21, 26, 43
Shakey, 261
SHOW-ARRAY, 58, 59, 61
SHOW-PIXELS, 197, 198, 198, 201
SHOW-QUEUE, 163
Shrdlu, 247, 261, 263
Simon, Herbert, 5
simplebase, 105
simulation techniques, 171
simulation time-step, 176
simulation, 6, 146, 153, 158, 164, 165, 179, 182, 188, 193
single-server queue, 145, 169
sketch, 197, 200, 201
slot, default, 217
slot, procedural attachment, 217
slot, reference, 217

solar system, 241
SOLVE-EQN, 90, 91, 98
sort array, 60, 63
sorting algorithm, 61
sorting, 6
space bar, 19
space probes, 194
space, 197
Spinoza, 257
sqroot, 65
square, 18, 197
standard deviation, 67
Stanford Research Institute, 118
statement, 122, 123, 124
statistical functions, 102
stochastic, 152, 161statistics, 6, 42, 43
stochastic process, 145
STORE, 105, 106, 107
string, 173, 174
STRIP-ZERO, 33, 63, 197
strong A.I., 261
structure, tree, 139
sublist, 8, 13-16, 19, 21, 24
subtotal, 55, 56
SWAP, 61
SYMBOL-PLIST, 102, 104, 132
SYMBOL, 8, 21
symbolic-expression, 8, 17, 18
system boundary, 87
system dynamics, 171, 193

T, 24, 26, 27, 27, 39, 41
table, 234, 238
tail, 9, 12
Taylor series, 160
TERPRI, 73, 100, 106
TEST-RAND, 158, 159
third, 46, 69
Thomas, Dylan, 249
time-series, 6, 84

TIME-STEP, 182, 190
times series, 71
TIMES, 15
top-down programming, 48
top-down structure, 100
tracks, 218
transformational grammar, 253
tree structure, 112, 113, 132, 139, 144, 169
tree, binary, 111, 115
tree, fill, 113
tree, search, 115

TREND-DEV, 75
TRUE, 26, 27, 28, 41
Turing Institute, 262
Turing machine, 261, 262
Turing test, 261, 262
Turing, Alan, 253, 261

unbound variable, 11, 24

undefined function, 13, 14
union, 124, 125
unit matrix, 90, 98
update, 124, 125
users, 11-13

value, seed, 161
values, 41
variable, 8-11, 21
variable, unbound, 11
variation, cyclical, 70
variation, random, 70
variation, seasonal, 70
vector, 57
vertex, 202, 203
VERTEXTP, 197, 201, 203, 204
vertices, 204, 215
Viking, 261
visual image, 196
von Neumann, John, 157

waiting, 145, 152, 154
Waltz algorithm, 263
weak A.I., 263
Weizenbaum, Joseph, 250
wend-while, 24
Williams, Bernard, 248
Winograd, Terry, 261, 263

zerop, 149, 169
zetalisp, 252

An Invitation

Sigma Press is still expanding–and not just in computing, for which we are best known. Our marketing is handled by John Wiley and Sons Ltd, the UK subsidiary of a major American publisher. With our speed of publication and Wiley's marketing skills, we can make a great success of your book on both sides of the Atlantic.

Currently, we are looking for new authors to help us to expand into many exciting areas, including:

Laboratory Automation
Communications
Electronics
Professional Computing
New Technology
Personal computing
Artificial Intelligence
General Science
Engineering Applications

If you have a practical turn of mind, combined with a flair for writing, why not put your talents to good use? For further information on how to make a success of your book, write to:

Graham Beech, Editor-in-Chief, Sigma Press,
1 South Oak Lane, Wilmslow, Cheshire SK9 6AR
or, phone 0625-531035

A practical, problem solving approach to LISP

Although one of the first of the 'high-level' languages, LISP has continued to be one of the most widely used. It has progressed beyond its acclaimed role in Artificial Intelligence and is now an excellent general purpose programming tool.

How to Solve It In LISP assumes no previous programming experience in any language and is particularly useful for those approaching computers for the first time. Essential LISP code is explained lucidly, with unique emphasis upon practical applications. LISP is shown to be ideal for problem-solving, and topics covered include statistics, data structures, simulation of biological systems, computer vision, pattern recognition, machine learning and expert systems.

The dialect employed is Golden Common LISP - a highly acclaimed standard version of LISP widely used on the IBM PC or compatibles such as the Amstrad PC range. However, the clear explanations of program algorithms, and detailed commentaries provided in the text, make the book essential reading for those wishing to program in any version of the language.

About the author

Patrick Hall has had a life-long obsession with computers a s, dating from an early exposure to SF films. He has since written a of books on computing, including *Amstrads and Artificial Intelligence*, a shed by Sigma Press.

He is a graduate of Fitzwilliam College, Cambridge and ght at schools in Leicester and Portsmouth, and now specialises in educa mputing.

KP-359-373

Published by

Sigma Press
1 South Oak Lane
Wilmslow
Cheshire, UK

ISBN 1 85058 005 7

Complete catalogue on request.
New authors are always welcome.

SIGMA PRESS